(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
28 June 2001 (28.06.2001)

PCT

(10) International Publication Number
WO 01/46834 A1

(51) International Patent Classification[7]: G06F 17/00, 7/00

(21) International Application Number: PCT/US00/34523

(22) International Filing Date: 19 December 2000 (19.12.2000)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
09/471,574 23 December 1999 (23.12.1999) US

(71) Applicant: 1ST DESK SYSTEMS, INC. [US/US]; 7 Industrial Park Road, Medway, MA 02053 (US).

(72) Inventors: GIBSON, Seann; 2696 Dayton Avenue, Columbus, OH 43202 (US). GIBSON, William, K.; 2201 Summit Street, Columbus, OH 43201 (US).

(74) Agents: ALTMAN, Gerald et al.; Morse, Altman & Martin, Suite 5F, 85 East India Row, Boston, MA 02110 (US).

(81) Designated States (national): AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CR, CZ, DE, DK, DM, DZ, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, UZ, VN, YU, ZA, ZW.

(84) Designated States (regional): ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

**Published:**
— *With international search report.*

*For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.*

WO 01/46834 A1

(54) Title: STREAMING METATREE DATA STRUCTURE FOR INDEXING INFORMATION IN A DATA BASE

(57) Abstract: A streaming metatree data storage structure in which each data item (40) is stored as a procession of nodes (52) and where data units common to more than one data item (40) are stored only once. The nodes (52) are stored in a logically linear fashion, or stream, and information (62) within a node (52) indicates its relationship to the other nodes and within the tree hierarchy. The data structure provides a mechanism for distributing the nodes (52) among multiple physical memory blocks (80) and for traversing backwards through the tree. Data items are added by creating a temporary tree for new items until it becomes too large, and then merging the temporary tree into the main tree.

# STREAMING METATREE DATA STRUCTURE FOR INDEXING INFORMATION IN A DATA BASE

## BACKGROUND OF THE INVENTION

### Field of the Invention

The present invention relates to storing information and, more particularly, to a tree configuration by which an indexing data base is stored in streaming memory and accessed.

### The Prior Art

In many computer applications, large amounts of information must be stored and accessed. Generally, during the process of deciding how this information is to be stored, a tradeoff must be made between time and memory. The time variable includes the amount of time necessary to store information, to locate a particular piece of information, and to recreate the information once located. The memory variable includes the amount of memory necessary to store the information and to store and execute the software necessary to store, locate, and recreate the information.

There are actually two time/memory issues related to storing information, the first issue being how the information itself is stored in an information data base and the second issue being how a particular item of information is found within the information data base. The simplest way to store information is linearly, that is, information is stored in data memory as it is received and is not modified or compressed in any way. In such a system, a given amount of information occupies a proportional amount of data memory. The main advantage of such a system is that the amount of time needed store the information is minimized. The main disadvantage is that data memory requirements and the time needed to retrieve the information grow in direct proportion to the amount of information stored.

The simplest way to find a particular item of information is to linearly search the entire information data base for

the item until it is found. This method is advantageous in that it is simple to implement, but the amount of time needed to find particular information is unpredictable in the extreme and the average time to find a particular piece of information can be unduly great.

An alternate method for finding information is to use a keyword data base, also called an index. The index is stored in memory separate from the information data base. Each keyword of the index includes a set of pointers that points to one or more locations in the information data base that correspond to that keyword. Thus, rather than searching a large information data base for particular items of data, an index is searched for keywords, typically greatly reducing the search time.

The simplest index structure is an alphabetic list of the data items, with each item followed by the appropriate pointers. The disadvantage of such a structure is that, in order to find any particular data item, the list must be searched from the beginning, leading to a potentially long search time. There are ways of decreasing the search time, such as by fixing the size of each index entry or by creating another list of pointers to each item in the index. Each increases the amount of memory needed, requiring a time/memory tradeoff.

An alternate structure for decreasing search time of an index data base is a tree structure, which consists of a group of related nodes, each node containing a subset of the stored data items, where the relationship between the nodes defines the data items. Each unique data item is stored as a set of linked nodes. In one tree structure, such as that described in U.S. Patent Nos. 5,488,717 and 5,737,732, common parts of data items are combined into single nodes, followed by nodes containing the unique parts of the data items. The node containing the first part of the data item is called the root node and is generally common to more than one data item.

The node containing the last part of the item is called the leaf node and is unique for each data item. The data base is searched from the root node for a known data item. When the search reaches the leaf node for that data item, a pointer or other identifier in the leaf node is used to locate the data item in the information data base.

The memory in which the index data base is stored has two forms, primary storage and secondary storage. Primary storage is typically the local random-access memory, or RAM. Secondary storage is typically a disk drive or other mass storage device. The significant differences between the two are that primary storage is much smaller and much faster than secondary storage. For example, current personal computers typically have 64 Mbytes of RAM and 10 Gbytes of disk storage space, a factor of 200 difference. Further, the time it takes to access the RAM is, on average, more than 100,000 times faster than to access the disk.

The typical index data base that is so large that linear storage is out of the question is also far too large to fit completely into primary storage. Consequently, most of the data base resides on disk, which means that the vast majority of search time is taken up by reading data from disk, not by the processing time needed to find the object of the search.

Additionally, most tree structures require that new data items be added individually, which means that the vast majority of inversion time, the process of adding data to a tree, is taken up by writing the updated tree to disk following each data item inversion. Typically, a trade-off must be made between inversion time and search time. When selecting a tree structure of the prior art, one must decide whether inversion time or search time is to be minimized because none of the tree structures of the prior art provide both fast inversion time and fast search time.

Thus, there continues to be a need for a data structure for indexes that provides for heavy concentration of data,

rapid and predictable information location times, rapid inversion times, and that is easily adapted to the physical structure of secondary storage media.

## SUMMARY OF THE INVENTION

An object of the present invention is to provide a data base structure that reduces secondary storage accesses during both the process of adding data items and searching for data items.

Another object is to provide a data base structure that minimizes secondary storage accesses while maximizing storage usage.

The essence of the streaming metatree (SMTree) of the present invention is its data storage structure. A typical prior art data structure uses randomly located nodes that point to the next succeeding node of the data item. Thus, nodes are followed by pointers to find a particular data item, jumping randomly between primary and secondary storage. On the other hand, the SMTree structure of the present invention stores nodes in a logically linear fashion, hence the term "streaming". Pointers are used in the present invention, although not in the same way as in the data structures of the prior art. Physical computer memory is composed of fixed-size blocks, throughout which the SMTree is distributed. Since the blocks can be randomly located in a physical sense, pointers are required. However, the data structure is such that a memory block will only be traversed at most one time during a search. Information stored within the node indicates its relationship to the other nodes and within the tree hierarchy. The SMTree structure is particularly suited for indexing-type structures.

There are two basic embodiments of the SMTree, a "horizontal" embodiment and a "vertical" embodiment, and a hybrid embodiment that uses components of both. The horizontal embodiment is most preferred because it is more efficient for the vast majority of applications.

Logically, the SMTree is composed of lists of alternative nodes. The root alternate list is a list of all data units that begin data items in the SMTree. When searching for a data item in the SMTree, the appropriate data unit from the root alternate tree is found and followed to the next lower level alternate list. This continues until the leaf node for the data item being search is reached. Following the leaf node is at least one identifier that references external objects.

Note that every data unit is considered to be part of an alternate list, and that many data units are in alternate lists that have only one member. In such a case, groups of single member alternate lists are combined into single nodes. For example, if the SMTree contains the two data items, "abbie" and "adamant", the lower level alternate list from the root alternate list member 'a' will have as members 'b' and 'd'. Logically, there will be three single member alternate lists following 'b', the first containing the member 'b', the second 'i', and the third 'e'. In order to save memory and processing time, the single member lists are combined into a node with the last member of an alternate list of more than one member. In this example, the nodes "bbie" and "damant" are the two members of the alternate list following 'a'.

In the horizontal embodiment, the nodes and identifiers are stored linearly and sequentially in memory, from the first member of the root alternate list to the leave node of the last data item, where the first physical sequence of nodes define the first data item of the SMTree. In order to determine where the nodes fit in the SMTree, each node has a header. The header includes (1) the size of the node so that it is known how many memory locations to skip if the node is not needed, (2) a flag indicating whether or not the node is a leave node so it is known that the search is at an end, (3) a flag indicating whether or not the node is the last node of

an alternate list, and (4) a flag indicating the presence of a memory block pointer. If the memory block pointer flag is set, the node is followed by a memory block pointer. The present invention includes a mechanism for dividing the SMTree into subtrees to fit into multiple memory blocks. This mechanism includes the memory block pointer flag, the memory block pointer, and a subtree header that includes a previous data unit value. When a memory block is jumped to, it must be determined which subtree in the block is the next to be traversed during the search. The previous data unit value contains the first data unit of the alternate list member node from which the jump took place. After the jump, these previous units are compared to determine the appropriate subtree to continue with. Implicit within this mechanism is that all subtrees in a block must be pointed to from members of the same alternate list, because this is the only way that the all of the previous units in a memory block can be guaranteed to be unique. On the positive side, it also guarantees that a block will only be traversed once during a search.

The present invention also provides two mechanisms for traversing backwards through the SMTree. In the first, each block has a head that includes a pointer to the higher-level block that contains the alternate list with pointers to the subtrees in the lower-level block. The subtree previous unit is used to determine which alternate list member points to the lower-level block. In the second method, each forwardly traversed block is pushed onto a stack, and are popped from the stack when traversing backwards.

The SMTree is traversed using the node header information. The search begins with finding the first unit of the data item to be search in the root alternate list. Nodes are skipped, using the node size value, to reach each member of the root alternate list until the matching member is found. The node in the stream following the matched node

is the first node of the next lower-level alternate list. The same procedure is followed with this alternate list as with the root alternate list. If a matching node is a leaf node, but the searched-for data item is not yet found, then the searched-for data item is not in the data base. After the searched-for data item is found, the identifier(s) that follow the leaf node are used as necessary.

Two mechanisms are contemplated for inversion, the process of adding data items to an SMTree. In the first, each new data item is added directly to the main SMTree. In the second, new data items are added a temporary SMTree until the temporary SMTree becomes too large, and then the temporary SMTree is merged with the main SMTree. Merging two SMTrees is a matter of merging the root alternate lists of the two SMTrees.

In the trees of the prior art, data items are added individually directly to the main tree, necessitating secondary storage accesses for each new data item. Since the SMTree of the present invention stores new data items in a temporary SMTree in fast primary storage before merging with the main SMTree in slower secondary storage, substantial savings in inversion time is realized. And the savings is accomplished without a corresponding increase in search time. The structure of the SMTree provides additional savings in inversion time by being particularly suited to being operated on by multiple processors. The alternate lists of the two SMTrees to be merged are divided between the processors at a point in the alternate list where the secondary storage between the two parts of the alternate list does not overlap. Since these different parts of an alternate list are completely independent of each other, the processors can operate completely independently.

The basic mechanism consists of traversing the SMTree searching for the new data item until a data unit of the existing data item differs from the new data item. Then a

new node containing the different unit and all subsequent units is added to the alternate list following the last common unit. The mechanism is actually much more complex due to the fact that the SMTree is spread among numerous memory blocks, so that the size of the subtrees must be taken into account. If a subtree becomes too large for the memory block, it must be split into two subtrees or moved to another memory block.

A data item is removed by searching for the leaf node of the data item to be removed, and removing nodes in reverse order up to and including the node of a multiple-unit alternate list. Optionally, any resulting single member alternate list can be compacted.

In summary, the present invention is a data structure adapted for use with a computer for storing data items in a data base, where each of the data items is composed of at least one data segment and each data segment is composed of at least one data unit including a first data unit. The data structure of the present invention comprises (a) a linear stream of nodes, each of the nodes containing a data segment, the nodes including predecessor nodes and successor nodes and being related by location in the stream; (b) progressions of nodes in the stream from a root alternate list node to a leaf node corresponding to progressions of data segments of the data items; (c) the progressions being traversed from node information included in each of the nodes, the node information including a size value indicating the number of data units in the node, a leaf node flag indicating if the node is a leaf node, and a terminal alternate flag indicating if the node is an alternate list terminal node; (d) each of the progressions being associated with at least one identifier that references at least one object external to the stream; (e) selected data segments of different data items in selected progressions of successor nodes being different for different data items; and (f) selected data

segments of different data items in selected predecessor nodes being common to said different data items.

Other objects of the present invention will become apparent in light of the following drawings and detailed description of the invention.

BRIEF DESCRIPTION OF THE DRAWINGS

For a fuller understanding of the nature and object of the present invention, reference is made to the accompanying drawings, wherein:

Fig. 1 shows a simple data base and corresponding index;

Fig. 2 is a prior art tree structure for the index of Fig. 1;

Fig. 3 is a block diagram of the hardware of a digital data processing system;

Fig. 4 is a block diagram of the software of a digital data processing system;

Fig. 5 is the tree of Fig. 2 restructured according to the horizontal embodiment of the present invention;

Fig. 6 is a portion of the tree of Fig. 5 with nodes incorporating an identifier;

Fig. 7 is a diagram of a node header;

Fig. 8 is a logical SMTree of the present invention for the index of Fig. 1;

Fig. 9 is a diagram of how the SMTree of Fig. 8 is stored in memory;

Fig. 10 is a diagram of how a single subtree in another block is referenced;

Fig. 11 is a diagram of how multiple subtrees in another block are referenced;

Fig. 12 is a diagram of a subtree header;

Fig. 13 is a diagram of a block header;

Fig. 14 is a diagram of how the SMTree of Fig. 11 is stored in memory;

Figs. 15 and 16 are diagrams showing how a data item is added to an SMTree;

Fig. 17 is a diagram of an SMTree to be merged into the SMTree of Fig. 5;

Fig. 18 and 19 are diagrams showing how the SMTree of Fig. 17 is merged into the SMTree of Fig. 5;

Fig. 20 is the tree of Fig. 2 restructured according to the vertical embodiment of the present invention; and

Fig. 21 is the tree of Fig. 2 restructured according to the hybrid embodiment of the present invention.

DETAILED DESCRIPTION

As an initial matter, this description of the present invention refers to a complete unique part of the information being stored as a data item, to a portion of the data item as a data segment, and to the smallest possible portion of a data item as a unit. In the typical information data base, the data will take the form of strings of characters, so that a data item is a string such as a sentence, a data segment is a substring (which can also be as large as the complete string or as small as a single character), and a data unit is a character. It is to be understood, however, that strings of characters are merely illustrative of a broad range of other data formats, such as binary numbers, single bits, and combinations of data formats. An example of the latter includes a word index for a text file where the first part of a data item is a word being indexed and the second part of the data item is a binary pointer to the location of the word in the text file.

<u>Index Trees</u>

Data is stored in data bases, of which there are a variety of organizations. The data base structure of the present invention is most suited to the index of an information data base. In an information data base/index combination organization 10, shown in Fig. 1, each unique data item 14 in the information data base 12 typically has a unique identifier 16 associated with it. This unique identifier 16 may be, for example, the location of the data

item 14 within the information data base 12. Each keyword 20 in the index 18 contains one or more identifiers 22, where each identifier 22 corresponds to an identifier 16 in the information data base 12. To find particular data items 14 of information, the index 18 is searched for keywords 20 and the associated identifiers 22 are used to locate the data items 14 in the information data base 12.

Data in a tree structure is stored so that successive duplicate segments are not duplicated in memory. For example, a typical prior art tree 30 containing the eight data items of Fig. 1, abbie, adamant, joe, joining, semester, stand, stanford, and stanley, is shown in Fig. 2. In this structure, each unit 32 is a character that is stored in a separate node 34. At the end of each data item is an additional node 36 holding the identifier 22 that associates the data item with the information data base.

When traversing the tree 30, a searcher first looks for a unit match in the left-most vertical group 38, denoted as an alternate list (the left-most alternate list is called the root alternate list), and follows the horizontal group 40 of the matched unit, as at 46. When the searcher encounters another alternate list 42, it again looks for a unit match and follows the corresponding horizontal group. This progression continues until the last node 44 of the data item is reached. The node 36 after the last node 44 of the data item contains the identifier 22 that points into the information data base.

## Computer Configuration

The graphic illustrations and pseudocode described below are implemented in a digital computation system as shown in Fig. 3. This system comprises an architecture including a central processing unit (CPU) 302, primary storage (local memory) 304, secondary storage (mass storage) 306, and input/output devices, including a keyboard 308, a mouse 310, and a display 312. Residing within the primary storage 304

as CPU-executable code is the operating system and user applications, as in Fig. 4. In the illustrated system, executable code 332 is generated from two inputs. The first input is the source code 322 written in the language chosen by the programmer and generated using a text editor 320. The source code 322 is processed by a compiler 324 into an intermediate code 326. The second input is a library of standard functions 328 that has previously been compiled. The intermediate code 326 and the library functions 328 are processed together by a linker 330 into machine-readable code 332, executable by the CPU 302, and stored in primary storage 304.

It is to be understood that the principles of the present invention are applicable to all digital computation systems. For example, the type of CPU 302 is not limited to a particular family of microprocessors, but can be chosen from any group of suitable processors. This is also true of the primary storage 304, secondary storage 306, and input/output devices 308, 310, 312. Likewise, the programmer may choose from a variety of programming languages, such as C, Pascal, BASIC, and FORTRAN, the appropriate compiler, linker, and libraries.

## Streaming Metatree

The essence of the streaming metatree (SMTree) is its data storage structure, which is different from any other previously implemented data storage structure. The typical tree structure is based on a node structure where each node has a pointer to the next succeeding node and the nodes are stored in random locations in memory. Thus, finding a particular data item is a matter of traversing the tree by following pointers to random memory locations, possibly jumping between primary storage and many different locations in secondary storage. On the other hand, the SMTree structure of the present invention is based on storing nodes in a logical order in memory, so that traversing through the

SMTree is a matter of reading through memory in a linear fashion, hence the term "streaming". This is not to say that pointers are not used in the present invention. As described in detail below, the SMTree is not physically a single stream of nodes. Physical computer memory, particularly secondary storage, is composed of fixed-size blocks. By necessity, the SMTree is distributed among these memory blocks, and since the blocks can be randomly located in a physical sense, pointers are required. However, nodes are stored in the memory blocks so that they are logically linear, that is, a memory block will be traversed only once, if at all, while searching for or retrieving a data item. Information stored within each node indicates the node's relationship to predecessor nodes and successor nodes in the stream, and its location in the hierarchy of the tree. The SMTree structure is used easily in information storage and retrieval applications, particularly for indexing-type structures.

There are two basic embodiments of the SMTree of the present invention, a "horizontal" embodiment and a "vertical" embodiment, where horizontal and vertical are referenced to the physical layout of Fig. 2. There is also a hybrid embodiment that uses components of both the horizontal and vertical embodiments.

At the end of each section of the horizontal embodiment, pseudocode implementing the subject discussed in the section is listed. Later pseudocode builds on the earlier pseudocode.

## Horizontal Embodiment

### 1. Structure of the Horizontal Embodiment

Note that when traversing forward through the tree of Fig. 2, some units 32 of a data item are no longer common to at least one data item, such as the vertical groups 'a'-'j'-'s' 38, 'b'-'d' 42, 'e'-'i', 'e'-'t', and 'd'-'f'-'l'. Where this occurs, a list of alternatives for the next unit is established and a selection must be made from this list as to

which path to take. Logically, all units can be considered as a member of an alternate list, some lists having only one member. In the horizontal embodiment, these single-unit alternate lists are compressed into segments 58 of more than one unit 54. Each segment 58 is contained in a node 52, resulting in a compressed tree 50, as in Fig. 5. Instead of 38 separate nodes 34, each containing one unit 32, there are only twelve nodes 52, shown in solid boxes, each containing a segment 58. Memory savings comes from not having to store overhead information for each of the units, but only for nodes. Multiple unit alternate lists 56 are shown in Fig. 5 by dashed boxes.

Following the last node of a data item, the leaf node, is the data item identifier 72, a special type of node. It is also contemplated that the data item identifier be incorporated into the leaf node, if such an arrangement is feasible, as at 48 in Fig. 6. If there is more than one identifier associated with a data item, they are preferably stored as an alternate list.

In the horizontal embodiment, the nodes 52 and identifiers 72 are stored linearly and sequentially in memory first from left to right and then from top to bottom. Consequently, a data item is stored as a forward progression of nodes 52 in memory, that is, each segment of a data item will be later in memory than its previous segment. This statement comes with caveats pertaining to memory blocks, as described below. In the example of Fig. 5, the nodes and identifiers are stored in the following order: [a], [bbie], [18], [damant], [11], [jo], [e], [56], [ining], [38], [s], [emester], [77], [tan], [d], [26], [ford], [63], [ley], and [0].

In order to determine where in the SMTree the nodes 52 fit, each node 52, along with the data segment component 60, includes a header 62, which contains the parameters of the node 52. As shown in Fig. 7, the header 62 includes (1) a

numeric field indicating the size of the node, (2) a binary (boolean) field, or flag, indicating whether or not the node is the end of a data item, (3) a flag indicating whether or not the node is the last node of an alternate list, and (4) a flag indicating the presence of a memory block pointer. Fig. 8 shows the tree 50 of Fig. 5 with the headers 62 for each node 52 and Fig. 9 shows how the nodes 52 are stored sequentially in memory. The first field 68 indicates the size of the node (SIZ field), that is, the number of units in the node. There are several ways to implement the size field 68. In order to maximize the size of a node, the preferred method is to assume that every node has at least one unit, and use the field to indicate the number of units more than the minimum size. For example, a node of five units will have a size field of 4, meaning that there are four more units in the node than the minimum of one. The second field 64 is the end of data item flag (EOI flag), where '1' indicates that the node is a terminal, or leaf, node, that is, at the end of the data item. The EOI flag 64 may also be used to indicate that additional information, such as the data item identifier 72, follows immediately after the node 52 or that the node 48 includes the data item identifier. The third field 66 is the terminal alternate flag (EAL flag), where '1' indicates that the node is the last node of an alternate list. The fourth field 70 is the next block pointer flag (NBP flag), which indicates whether a block pointer follows the node.

As indicated above, it is expected that the SMTree will reside in two types of storage, primary storage (RAM) and secondary storage (disk). Disks store information in fixed-length records, which are typically from 1024 to 4096 bytes long. The vast majority of SMTrees cannot fit into one record, so the present invention includes a mechanism for dividing the SMTree into subtrees to fit into multiple blocks. The present invention contemplates that the block

size is not dependent upon the physical constraints of the disk drive. However, the block size is preferably chosen for efficiency, and will typically be the size of a disk record. The NBP flag 70 indicates whether or not a pointer to another block follows the node. In the preferred implementation, a single bit is used for the NBP flag 70, which means that all block pointers must be the same size. It is also contemplated that more bits can be used as the NBP flag 70 so that block pointers of different sizes can be used.

Fig. 10 shows the SMTree of Fig. 8 split between two blocks. The root alternate list 76 of the SMTree is in a root block 80 and a lower-level subtree 78 in an another block 82. A block pointer 74 following the "a" node points to the subtree block 82. Note that the subtree 78 always begins with an alternate list.

Implicit within the structure of the present invention is that a subtree cannot be larger than a block. However, a subtree can be smaller than a block so that it is possible to store more than one subtree in a block, increasing memory efficiency, as in Fig. 11. In order to maximize the number of blocks that can be used by the SMTree, the NBP field 74 points to a block, not to the subtree within the block. Consequently, a means is needed to find the appropriate subtree within the block. To facilitate such a means, each subtree includes a subtree header 84 that contains two fields, a previous unit field (PRU field) 86 and a subtree length field (STL field) 88, as shown in Fig. 12. The PRU field 86 contains a copy of the first unit of the node that preceded the jump to the subtree block, which is guaranteed to be unique for all subtrees in a block. A consequence of using the previous unit to find the appropriate subtree within a block is that all subtrees within a block must be pointed to by members of the same alternate list. Because the previous unit is used to differentiate subtrees within a block, the previous unit must be unique for all subtrees of

the block. And the only way to guarantee that all previous units are unique is to require that they all be members of the same alternate list; all members of an alternate list are unique, there are no duplicates.

This requirement has a positive consequence: that any block will only be traversed once while searching for or retrieving a data item. Since an alternate list, by definition, contains only one data segment of a single data item, and all subtrees in a block are pointed to from the same alternate list, only one subtree in a block will be traversed during a single data item search. Thus, a memory block will only be traversed once during a data item search.

Generally, it is necessary to have the ability to traverse back through the SMTree. The present invention contemplates at least two methods for doing so. In the first, each block includes a header, shown in Fig. 13, with a previous block pointer field 106 (PBP field). In order to find the alternate list node that jumped to the lower-level subtree 96, the PBP field 106 contains a pointer to the block 90 that holds the higher-level subtree 94 containing the alternate list. The fact that the PBP field 106 only points to the higher-level block 90 implies the restriction that a jump to another block can only be performed from the root alternate list of a subtree. If a jump to another block occurred from any other alternate list, there would be no way of traversing forward through the higher-level subtree to find the alternate list that was the source of the jump to the lower-level block 92. The consequence of this restriction is that a single PBP field 106 is all that is needed to traverse backwards.

The second method contemplated for traversing backwards through the SMTree is to push each block that is traversed in the forward direction onto a block history stack. Alternatively, enough information is pushed on the stack so that the block and the subtree within the block can be

located, including, for example, a pointer to the block within the entire SMTree and an index into the block for the subtree. In traversing backwards, each item popped from the stack is or indicates the next higher subtree in the data item hierarchy. This is the preferred method, and is the one that is implemented in the pseudocode.

The following pseudocode defines the data structures for the operational pseudocode routines shown later.

---

```
let kMaxBlkLen define CONSTANT M
// where M is the maximum length of an mtBlock.
```

---

```
let kMaxNodeLen define CONSTANT N
// where N is the maximum length of an mtNode.
```

---

```
// mtBlockPointer
let mtBlockPointer define B
// where B is an integer value defining the location of a
// block in storage.
```

---

```
let SMTree define S
// Where S encompasses the storage of the SMTree.
```

---

```
// mtUnit
// The basic atomic unit of an alternate list. An example
// would be a single ASCII character.
let mtUnit define C
// where C is a single unit of an alternate list.
```

---

```
// mtNodeHeader
// Information related to an mtNode.
let mtNodeHeader define SIZ,EAL,EOI,NBP
// where SIZ defines the node size: a value from 0 to k
//      corresponding to the number of single item alternate
//      lists in the node.
// where EAL defines the alternate list terminal: one of
//      TRUE or FALSE. If TRUE, the first unit of the
//      mtNode is the last alternate in its respective
```

// alternate list.<br>
// where EOI describes the next mtNode: one of TRUE or<br>
// FALSE. If TRUE, this mtNode is a leaf node and this<br>
// data item has ended. If FALSE, the next mtNode is a<br>
// continuation of this data item.<br>
// where NBP indicates a block pointer at the end of this<br>
// mtNode: one of TRUE or FALSE.

---

// mtNode<br>
// The mtNode defines a series of alternate lists in an<br>
// mtBlock. $C_0$ is the first unit and is always a member of<br>
// a multiple unit alternate list. All units which follow<br>
// $C_0$ are in logically separate single unit alternate lists<br>
// (defined by $C_n$). This precludes the need for each<br>
// alternate item to have its own header. Following the<br>
// series, there may be a continuation of the data item of<br>
// which the series is a part (denoted by Hdr.EOI = FALSE)<br>
// or the series may terminate the data item (denoted by<br>
// Hdr.EOI = TRUE) signaling that the next mtNode.$C_0$ is the<br>
// next unit in the alternate list of which $C_0$ is a part (if<br>
// Hdr.EAL is FALSE) or that the next mtNode.$C_0$ is from a<br>
// previous alternate list (if Hdr.EAL is TRUE).<br>
let mtNode define Hdr,($C_0$,$C_1$,...,$C_k$),BlkPtr<br>
// where Hdr is an mtNodeHeader.<br>
// where ($C_0$,$C_1$,...,$C_k$) is a series of k+1 mtUnits in an<br>
// mtBlock denoting units in connected alternate<br>
// lists, where k+1 <= kMaxNodeLen. $C_0$ is unique in<br>
// that it is part of a multiple unit alternate list.<br>
// ($C_1$,$C_2$,...,$C_k$) are units in single unit alternate<br>
// lists only.<br>
// where BlkPtr is a pointer to an mtBlock: one of<br>
// mtBlockPointer or NULL.

---

// mtAlternateList<br>
// This alternate list is not used explicitly. Its<br>
// inclusion here is to display the logical layout of an<br>
// alternate list in the SMTree. Such a structure could be

```
// initialized by perusing related mtNodes in a block and
// collecting the C0 character of each one until the
// terminal alternate (defined as Nx.Hdr.EAL = TRUE).
let mtAlternateList define NumUnits,(N1.C0,N2.C0,...,Nk.C0)
// where NumUnits is the number of units in the alternate
//      list.
// where Nx.C0 is the first unit of an mtNode and
//      N1.C0,N2.C0,...,Nk.C0 defines a group of units in
//      sorted order comprising one complete alternate
//      list.
```

---

```
// mtSubtree
// A collection of related mtNodes stored in contiguous
// order.
let mtSubtree define PRU,STL,(N1,N2,...,Nk)
// where PRU is an mtUnit in the root alternate list of
//      the subtree traversed to reach this subtree.
// where STL is the cumulative length of all mtNodes
//      (N1,N2,...,Nk) in the subtree.
// where Nx is an mtNode related linearly to other nodes in
//      the subtree.  N1.C0 always defines the first
//      alternate unit of the first alternate list (root) of
//      a block for the subtree.  Nk.Ck always defines the
//      last alternate unit of the last alternate list
//      (leaf) of the subtree.
```

---

```
// mtBlock
// A constant length contiguous stream of data in storage.
// Holds one or more mtSubtrees.
let mtBlock define BKL,(S1,S2,...,Sk)
// where BKL is the block length up to length kMaxBlkLen.
// where Sx is an mtSubtree.
```

---

```
// mtBlockInfo
// Used exclusively in the mtBlockInfoStack to reference one
// of a chain of blocks during a particular traversing of an
// SMTree.  This allows the code to traverse backwards
```

```
// through the SMTree.
let mtBlockInfo define BlkPtr,s
// where BlkPtr is a pointer to an mtBlock.
// where s is an integer index to the current subtree in the
//      block.
```

---

```
// mtBlockInfoStack
// A stack which references the chain of blocks that were
// traversed to get to a particular point in the SMTree.
// This allows the code to traverse backwards through the
// SMTree.
let mtBlockInfoStack define top,(B1,B2,...,Bk)
// where top defines the location of the top of the stack.
// where B is an mtBlockInfo.
```

---

```
// mtPosition
// mtPosition is the main object passed into most routines.
// It keeps track of the current position in the entire
// SMTree.
let mtPosition define Blk,BlkStk,Sub,Node,u,(A1,A2,...,Ak),a
// where Blk is the current mtBlock.
// where BlkStk is an mtBlockInfoStack.
// where Sub is the current mtSubtree in the block
//      referenced by Blk.
// where Node is the current mtNode in the mtSubtree
//      referenced by Sub.
// where u is an integer index to the current mtUnit in the
//      mtNode referenced by Node.
// where A is an mtUnit and (A1,A2,...,Ak) defines the total
/       accumulated units up to the current position in the
//      SMTree.
// where a is an integer index to the current mtUnit in
//      (A1,A2,...,Ak).
```

---

2. Traversing Forward in the Horizontal Embodiment

The SMTree of Fig. 9 is traversed using the information in the headers 62. For example, suppose one wishes to find

the search word "joining" in the SMTree. The search begins at the first node of the root alternate list 200, the "a" node 202. Since the "a" node 202 does not start with the unit "j", the first unit of the search word, the search must find the next node of the root alternate list 200. It is known that there are additional nodes in the root alternate list 200 because the EAL flag 66 is '0'. Each successive node after the "a" node 202 must be read and discarded until the next node, in this case the "jo" node 212, of the root alternate list 200 is found. The "bbie" node 204 is read first. The location of the beginning of the "bbie" node 204 is calculated as a displacement from the "a" node 202 using the SIZ field 68 of the "a" node 202 and the fixed size of the header 62. It is known that the "bbie" node 204 is not the next root alternate list node because the EOI flag 64 of the "a" node 202 is not set, meaning that there is at least one horizontal node from the "a" node 202. It is also known that the "bbie" node 204 is the first node of a second level alternate list 206 because the structure of the SMTree requires that all nodes be members of an alternate list. It is further known that the "bbie" node 204 is not the end of its alternate list 206 because the EAL flag 66 is not set.

The size of index data bases leads inevitably to many levels of alternate lists. In the present example, the SMTree has three levels, but it can easily be seen that the number of levels is limited only by the length of the longest data item. Thus, some mechanism for keeping track of the alternate list level that is currently being traversed is needed. The structure of the SMTree lends itself to the use of a counter that is initialized to "1" at the beginning the current alternate list. The counter is incremented for any subsequent node that is not a leaf node, and is decremented for any node that is the end of an alternate list. When the counter equals zero, the node following the current node in

the stream is the next alternate of the alternate list of which the counter is keeping track.

In the current example, reading the "a" node 202 causes the alternate list counter to be initialized to "1" to indicate that this is the alternate list that the first unit of the search word is in. Reading the "bbie" node 204 causes the counter to remain unchanged because it is a leaf node and is not at the end of an alternate list.

Since it is already known that that the "bbie" node 204 is not of interest to the search, the next node 210, "damant", is read. Its location is determined in essentially the same way as the "bbie" node 204. Its offset from the "bbie" node 204 is the sum of the SIZ field 68 of the "bbie" node 204, the "bbie" node header 62, and the length of the data item identifier 208 that follows the "bbie" node 204. It is known that the "damant" node 210 is the next node of the second level alternate list 206 because the EOI flag 64 of the "bbie" node 204 is set, meaning that there are no more horizontal nodes following the "bbie" node 204. The EAL flag 66 of the "damant" node 210 indicates that it is the last node of an alternate list 206, so the alternate list counter is decremented by "1" to "0".

Again, since the "damant" node 210 is not of interest to the search, the next node 212, "jo" is read. Because the alternate list counter is now "0", the "jo" node 212 is the next member of the root alternate list 200. A comparison is performed and it is found that the units of the "jo" node 212 are the same as the first two units of the search word "joining".

Since the EOI flag 64 of the "jo" node 212 is not set, the next node 214, "e", is read, starting a new alternate list level. A comparison shows that this is not the next unit of the search word, so it is discarded. Because of the "e" node 214 flag settings, it is known that the next node 216, "ining", is the next member of the alternate list. That

node 216 is read, compared, and found to complete the search word. Finally, the data item identifier 218 following the "ining" node 216 is read and used as desired.

Note that Fig. 9 represents data in a single block. As indicated, an SMTree may exist over several blocks, as in Figs. 10 and 11. Fig. 14 shows how the SMTree of Fig. 11 is stored in memory. In this case, a search for the data item "joining" would need to jump from Block A 90 to block B 92. Note that block B 92 has two subtrees 96, 98 and that the second subtree 98 includes the continuation of the data item "joining". The NBP field 232 following the "jo" node 230 will cause the jump to block B 92. Prior to the jump, information for the current block is pushed on the block history stack. After the jump, the PRU field 96 of the subtree headers are checked, in sequence, until the matching unit "j" is found. The first subtree header 234 has a PRU field 96 of "a", so it is discarded. The next subtree header 236, which is reached by using the STL field 88 of the previous subtree header 234, has a PRU field 96 of "j", meaning that it is the correct subtree. Then the subtree 98 is traversed as described above.

Example pseudocode for traversing forward to retrieve a data item from an SMTree follows. The last several routines, FindDataItem, FindPartialDataItem, and RetrieveDataItem, are the routines that would be accessed by an external program to retrieve a data item from the SMTree.

---

```
// GetFirstBlockPosition
// Make a new mtPosition and set it to the first unit in the
// first node of the first subtree in the given Block.
// Note: this mtPosition will not have a block info stack so
// it will not be possible to traverse backward in the tree
// from this point.
GetFirstBlockPosition(mtBlock Block)
    declare mtPosition pos
    pos.Blk = Block
```

```
    if (GET_NUM_SUBTREES(Block) > 0)
        pos.Sub = Block.S1
        if (GET_NODE_COUNT(pos.Sub) > 0)
            pos.Node = Block.S1.N1
        else
            pos.Node = NULL
        end if
    else
        pos.Sub = NULL
        pos.Node = NULL
    end if
    pos.u = 0
    return(pos)
end GetFirstBlockPosition
```

---

```
// GetFirstPosition
// Return a position object pointing to the first unit
// in the root block of the given SMTree.
GetFirstPosition(SMTree S)
    declare mtBlock B
    B = GET_ROOT_BLOCK(S)
    declare mtPosition pos
    pos = GetFirstBlockPosition(B)
    return(pos)
end GetFirstPosition
```

---

```
// GoNextBlock
// Using a given block pointer, read a block from storage
// into the given mtPosition pos.  Search the mtSubtrees in
// the block to find the subtree corresponding to FromUnit.
GoNextBlock(mtPosition pos,mtBlockPointer BlkPtr,
            mtUnit FromUnit)
    // Gather information about the current block in order
    // to push it onto the stack in pos.  This is necessary
    // to later traverse backwards through the SMTree.
    declare mtBlockInfo BkInfo
    // Get the pointer reference of the current block.
```

```
    BkInfo.BlkPtr = POINTER(pos.Blk)
    // Store the current subtree index.
    BkInfo.s = GET_SUBTREE_INDEX(pos.Sub)
    // Push the mtBlockInfo onto the mtBlockInfoStack in
    // pos.
    PUSH(pos.BlkStk,BkInfo)
    // Read the new block from storage.
    declare mtBlock B
    B = READ_BLOCK(GET_SMTREE_FOR_BLOCK(pos.Blk),BlkPtr)
    // Set B in the mtPosition to that block.
    pos.Blk = B
    i = 1
    // Loop through all the subtrees until PRU corresponds
    // to FromUnit.  This is the correct subtree.
    loop while B.Si.PRU <> FromUnit
        i = i + 1
    end loop
    // Set the other mtPosition variables to their initial
    // values.
    pos.Sub = B.Si
    pos.Node = B.Si.N1
    pos.u = 0
    // Return the new position.
    return (pos)
end GoNextBlock
```

---

```
// GoNextNode
// From the given mtPosition, go to the next mtNode.
GoNextNode(mtPosition pos)
    // Get the index position of the current mtNode.
    x = GET_NODE_INDEX(pos.Node)
    // Increment the position by one.
    x = x + 1
    // Set N to the next mtNode.
    pos.Node = pos.Sub.Nx
    // Set unit counter to index the first unit.
```

```
    pos.u = 0
    return (pos)
end GoNextNode
```

---

```
// ScanToLastNodeFromPosition
// From the given mtPosition, scan forward in the tree until
// reaching the node just before the node that represents
// the next alternate unit.  This is used by other routines
// to locate the next alternate unit by perusing the node
// tree.
ScanToLastNodeFromPosition(mtPosition pos)
    // If the next node is a continuation...
    if (pos.Node.Hdr.EOI = FALSE)
        counter = 1
        // Loop that 'counts' nodes off until pos reaches
        // the node just before the one with the next
        // alternate unit.  Because of the implicit nature
        // of the SMTree, counter will reach zero at the
        // correct location.
        loop
            GoNextNode(pos)
            // If the node contains the last alternate item
            // in the alternate list, subtract from counter.
            if (pos.Node.Hdr.EAL = TRUE)
                counter = counter - 1
            end if
            // If the next node continues the data item,
            // add one to counter (notice this is not an
            // else case.  Both ifs can execute).
            if (pos.Node.Hdr.EOI  = FALSE)
                counter = counter + 1
            end if
        // In correct position to read next unit of
        // alternate list when counter reaches zero.
        until counter = 0
    end if
```

```
    return (pos)
end ScanToLastNodeFromPosition
```

---

```
// GoNextAlternateUnit
// Find the next alternate unit in an alternate list.
// Because of the structure of the SMTree, this requires a
// complex traversing of successive mtNodes.  Each node from
// the position pos must be traversed in linear order and
// checked to determine if it contains the next alternate
// unit.  If pos is on an alternate unit consisting of only
// one unit, then this function will return FALSE.
GoNextAlternateUnit(mtPosition pos)
    // First check to see if this is the last alternate
    // in the current alternate list.
    if (pos.u > 0 OR pos.Node.Hdr.EAL = TRUE)
        // If pos.u is greater than zero, the alternate list
        // only consists of one unit.  This is so because C0
        // of an mtNode defines itself as part of a multiple
        // unit alternate list.  Each Cx after C0 is defined
        // as a single unit alternate list only.  Return
        // FALSE to denote that there was no next alternate
        // unit.
        return (pos,FALSE)
    else // Otherwise search for the next alternate unit.
        // Find the last node derived from the current
        // alternate unit.
        ScanToLastNodeFromPosition(pos)
        // GoNextNode will return the next alternate unit
        // in pos.Node.C0.
        GoNextNode(pos)
        // set pos.u to the first unit in the node.
        pos.u = 0
        // Replace the current unit in the tracking data
        // item with the first unit of the new node.
        x = pos.a
        pos.Ax = pos.Node.C0
```

```
        // Return the new position with TRUE result.
        return (pos,TRUE)
    end if
end GoNextAlternateUnit
```

---

```
// FindAlternateUnit
// If a given alternate list contains an alternate unit R,
// return TRUE with the position i of R, else return FALSE.
// The alternate list in question is determined by the pth
// item of StartNode (an mtNode).
FindAlternateUnit(mtUnit R,mtPosition pos)
    // An alternate list contains R if one alternate in the
    // list is R.  Loop through the alternate units and
    // return TRUE if R is matched.  If R is not found and
    // the last alternate unit is reached, return FALSE.
    loop
        x = pos.u
        if (pos.Node.Cx = R)
            return (pos, TRUE)
        end if
    // The loop will bug out if pos has gone beyond where R
    // should be in a sorted list or GoNextAlternateUnit
    // returns FALSE (meaning there are no more alternates).
    until (pos.Node.Cx > R OR
            GoNextAlternateUnit(pos) = FALSE)
    return (pos,FALSE)
end FindAlternateUnit
```

---

```
// IsLastAlternate
// Returns TRUE if the current alternate in the given
// mtPosition is the last alternate unit in it's respective
// alternate list.
IsLastAlternate(mtPosition pos)
    // If the current unit is the last unit in the node AND
    // the next node is not a continuation of the current
    // node, then this is the last alternate.
    return (pos.u = pos.Node.Hdr.SIZ AND
```

```
        pos.Node.Hdr.EOI = TRUE)
end IsLastAlternate
```

---

```
// GoNextAlternateList
// Set the given mtPosition pos to the beginning of the next
// alternate list in the data item.
GoNextAlternateList(mtPosition pos)
    // If not on the last unit in the node...
    if (pos.u < pos.Node.Hdr.SIZ)
        // Increment the unit in the current position.
        // This is automatically the next alternate list
        // and will consist of only one unit.
        pos.u = pos.u + 1
    else // The current unit is the last one in the node.
        // If the following node is the next alternate
        // list...
        if (pos.Node.Hdr.EOI = FALSE)
            // If a pointer to another block follows...
            if (pos.Node.Hdr.NBP = TRUE)
                // Read the new block into the current
                // position.
                GoNextBlock(pos,pos.Node.BlkPtr,pos.Node.C0)
            else
                // Go to the next sequential node which will
                // be the location of the next alternate
                // list.
                GoNextNode(pos)
            end if
        else
            // This is the last alternate list in the data
            // item, so return FALSE.
            return (pos,FALSE)
        end if
    end if
    // Add this unit to the tracking data item in pos.
    pos.a = pos.a + 1
```

```
    x = pos.a
    y = pos.u
    pos.A_x = pos.Node.C_y
    // Return the amended position with TRUE.
    return (pos,TRUE)
end GoNextAlternateList
```

---

```
// FindFromPosition
// Find the data item S_1...S_j starting from the current
// position in the tree, returning TRUE if it exists and
// FALSE if it does not.
FindFromPosition(mtPosition pos,mtUnit S_1...S_j)
    if (pos.Node <> NULL)
        n = 1
        // While not at the end of the data item, S_n has been
        // found in the current alternate list, and not at
        // the last alternate item.
        loop while (n < j AND FindAlternateUnit(S_n,pos) AND
                        GoNextAlternateList(pos))
            n = n + 1
        end loop
        // Return TRUE if at the end of S and S_j is in the
        // alternate list and the alternate list is the last
        // alternate list in the branch.
        return (pos, (n = j AND FindAlternateUnit(S_n,pos) AND
                        IsLastAlternate(pos))
    else
        // If there are no nodes then the tree is completely
        // empty so the search returns FALSE.
        return (pos, FALSE)
    end if
end FindFromPosition
```

---

```
FindDataItem(mtPosition pos,SMTree SMT,mtUnit S_1...S_j)
    // Go to the first node of the SMTree.
    pos = GetFirstPosition(SMT)
    // The substance of the routine exists in
```

```
    // FindFromPosition.
    return (FindFromPosition(pos,S1...Sj))
end FindDataItem
```

---

```
// FindPartialDataItem
// Find the data item S1...Sj in SMT, returning TRUE if it
// exists as part of or all of a data item in the index.
// Return FALSE if the partial data item does not exist.
FindPartialDataItem(mtPosition pos,SMTree SMT,
                            mtUnit S1...Sj)
    // Go to the first node of the SMTree.
    pos = GetFirstPosition(SMT)
    n = 1
    // While not at the end of the data item, Sn has been
    // found in the current alternate list, and not beyond
    // the last alternate item.
    loop while (n <= j AND FindAlternateUnit(Sn,pos) AND
                    GoNextAlternateList(pos))
        n = n + 1
    end loop
    // Return TRUE if at the end of S and Sj is in the
    // alternate list.  Return FALSE if the search
    // terminated prematurely.  pos will be set to the
    // last alternate list which matched.
    return (pos,(n = j))
end FindPartialDataItem
```

---

```
// RetrieveDataItem
// From the current position, retrieve the first data item
// by scanning along the first unit of each successive
// alternate list.
RetrieveDataItem(mtPosition pos)
    // Loop through each alternate list until on a leaf.
    loop while (GoNextAlternateList(pos) = TRUE)
    end loop
    return (pos)
end RetrieveDataItem
```

3. Traversing Backward in the Horizontal Embodiment

Traversing backwards through the SMTree is typically used when an ordered list of data items in the SMTree is needed, such as a display of an alphabetical list of the data items. After the first data item is found, the SMTree is traversed backward to find the next higher alternate list from the leaf node of the first data item. Then the SMTree is traversed forward from the next unit of the alternate list to find the next data item. In this way, a sorted list of data items is generated without having to start from the root alternate list for each data item, saving a significant amount of processing time and disk accesses.

As indicated above, the preferred method for traversing backward is to store the forward traverse history of the current data item by node. Traversing backward through the SMTree is then a matter of reading the forward traverse history stored in the information kept for the current node. If the prior node is within the same block, the node is taken from the history information. If the prior node is in another block, the prior block information is popped from the block history stack described above. Then the root alternate list of the higher-level subtree 94 is traversed to find the node having the first unit that matches the PRU field 86 of the lower-level subtree 96.

Example pseudocode for traversing backward through the SMTree follows. The last routine, RetrieveNextDataItem, is the routine that would be iteratively accessed by an external program to retrieve a sorted list of data items.

---

```
// GoPreviousBlock
// Change pos to reflect a new position in the tree by going
// back to the previous block. To do this, pop the block
// info stack and use the information to find both the block
// and the correct subtree.
```

```
GoPreviousBlock(mtPosition pos)
    // Pop the block info stack in pos to get information
    // about the previous block.
    declare mtBlockInfo BkInfo
    BkInfo = POP(pos.BlkStk)
    // Read the new block from storage.
    declare mtBlock B
    B = READ_BLOCK(GET_SMTREE_FOR_BLOCK(pos.Blk),
                    BkInfo.BlkPtr)
    // Set B in the mtPosition to that block.
    pos.Blk = B
    // Set the subtree of pos to the subtree stored in the
    // stack.
    i = BkInfo.s
    pos.Sub = B.S_i
    // Set the other mtPosition variables to their initial
    // values.
    pos.Node = B.S_i.N_1
    pos.u = 0
    // Return the new position.
    return (pos)
end GoPreviousBlock
```

---

```
// GoPreviousNode
// From the given mtPosition, go to the previous mtNode.
GoPreviousNode(mtPosition pos)
    // Get the index of the current node
    x = GET_NODE_INDEX(pos.Node)
    // If on the very first mtNode...
    If (x = 1)
        // Return FALSE to indicate no more nodes in the
        // block.
        return (pos,FALSE)
    else
        // Decrement the count of the current mtNode.
        x = x - 1
```

```
        // Set N to the previous mtNode.
        pos.Node = pos.Sub.Nx
        // Set unit counter to index the first unit.
        pos.u = 0
        return (pos,TRUE)
    end if
end GoPreviousNode
```

---

```
GoPreviousAlternateUnit(mtPosition pos)
    // Get the index of the current node.
    x = GET_NODE_INDEX(pos.Node)
    // Check to see if there is a previous alternate unit.
    if (pos.u > 0 OR x = 1)
        // If pos.u is greater than zero, then pos currently
        // references a single unit alternate list, OR there
        // is no previous node, then this is the first
        // alternate unit - so return FALSE.
        return (pos,FALSE)
    else
        // First check the previous node without going to it
        // to see if the current node is the first in its
        // respective alternate list.
        x = x - 1
        if (pos.Sub.Nx.Hdr.EOI = FALSE AND
                    pos.Sub.Nx.Hdr.NBP = FALSE)
            // The node previous to the current node is not
            // a leaf node.  This indicates that the
            // alternate unit was the first in the alternate
            // list, so return FALSE.
            return (pos,FALSE)
        else    // There is a previous alternate unit, so
                // attempt to find it.
            // Go to the previous node first.
            GoPreviousNode(pos)
            // Now immediately check to see if it is marked
            // as a terminal alternate.  If the first
```

```
        // previous node is marked as a terminal
        // alternate, then further checking is
        // necessary.  Otherwise the structure dictates
        // that pos.Node.C0 is the previous alternate
        // unit.  So simply skip further testing and
        // return with this previous node.
        if (pos.Node.Hdr.EAL = TRUE)
            // counter keeps track of alternate lists.
            counter = 1
            // Loop through previous nodes until we find
            // the previous alternate unit.  This loop
            // 'counts' nodes off until pos reaches the
            // node which holds the previous alternate
            // unit.  Because of the implicit nature of
            // the SMTree, counter will reach zero when
            // the correct node is found.
            loop while (counter > 0)
                // Get the previous node.
                GoPreviousNode(pos)
                // If this node contains the last
                // alternate item, increment counter.
                if (pos.Node.Hdr.EAL = TRUE)
                    counter = counter + 1
                end if
                // Notice no 'else' case.  Both ifs can
                // execute.  If this node is not a leaf
                // node, decrement counter.
                if (pos.Node.Hdr.EOI = FALSE)
                    counter = counter - 1
                end if
            end loop
        end if
        // Return the new position with TRUE.
        return (pos,TRUE)
    end if
```

```
    end if
end GoPreviousAlternateUnit
```

---

```
// GoPreviousAlternateList
// Set the given mtPosition pos to the first unit of the
// previous alternate list in the chain.
GoPreviousAlternateList(mtPosition pos)
    // If pos.u is greater than zero, then pos currently
    // references a single unit alternate list.
    if (pos.u > 0)
        // Simply decrement the index to the mtNode to get
        // to the previous alternate item.
        pos.u = pos.u - 1
    else // pos currently references first unit of node.
        // Must now search backwards in the SMTree, mtNode
        // by mtNode to find the previous alternate list.
        // First, back up to the first mtUnit in the current
        // alternate list.
        loop while (GoPreviousAlternateUnit(pos))
        end loop
        // Once here, must discover if the previous mtNode
        // is in the current block or the previous block.
        x = GET_NODE_INDEX(pos.Node)
        // If the current mtNode is the first one in
        // the subtree...
        if (x = 1)
            // Get the PRU of the previous block before
            // changing pos.
            P = pos.Sub.PRU
            // Go to the previous block.
            GoPreviousBlock(pos)
            // Now find the origin mtUnit in the root
            // alternate list of the previous block.
            FindAlternateUnit(P,pos)
            // Finally, trace down to the end of the current
            // mtNode.  This will leave pos in the exact
```

```
            // spot of the alternate list previous to the
            // original mtPosition passed in pos.
            pos.u = pos.Node.Hdr.SIZ
        else // The current mtNode is still in the block.
            // Get the previous node.
            GoPreviousNode(pos)
            // Increment pos.u to the end of the node.  This
            // will be the previous alternate list.
            pos.u = pos.Node.Hdr.SIZ
        end if
    end if
    // Subtract from the tracking string in pos.
    pos.a = pos.a - 1
    // Return the amended position with TRUE.
    return (pos,TRUE)
end GoPreviousAlternateList
```

---

```
// RetrieveNextDataItem
// From the current position, retrieve the next data item in
// sorted order by backing up alternate lists to the next
// fork, then traversing forward.
RetrieveNextDataItem(mtPosition pos)
    // For as long as no 'next' alternate unit exists, keep
    // backing up to the previous alternate list.
    loop while (GoNextAlternateUnit(pos) = FALSE)
        // GoPreviousAlternateList will return FALSE
        // if there is not a previous list, i.e., the end of
        // the root alternate list of the SMTree has been
        // reached.
        if (NOT GoPreviousAlternateList(pos))
            // No more data items exist so return FALSE.
            return (pos,FALSE)
        end if
    end loop
    // Exiting the loop means a position in the SMTree has
    // been reached from where the next data item can be
```

```
    // retrieved.  Now simply follow traverse forward to get
    // the data item.
    RetrieveDataItem(pos)
    return (pos,TRUE)
end RetrieveNextDataItem
```

---

4. Adding Data Items to the Horizontal Embodiment

Generally during the life of an indexing data base, data items must be added, a process known as inversion. In the trees of the prior art, new data items must be added one at a time to the main tree, necessitating secondary storage accesses for each new data item. One object of the present invention is to minimize reads from and writes to secondary storage because of the times involved when compared to the actual processing of the SMTree. To this end, one optional aspect of the present invention is that the SMTree inversion process begins by combining all new data items into a temporary SMTree in primary storage until the temporary SMTree becomes too large or unwieldy for primary storage. Only at that time is the temporary SMTree merged with the main SMTree. Thus, in the SMTree of the present invention no secondary storage accesses are made until the temporary SMTree is merged with the main SMTree. Since dozens, and perhaps hundreds or thousands, of data items may be held in the temporary SMTree, it is obvious that the present invention provides a substantial savings in the time needed to add data items to the SMTree. And, unlike the data structures of the prior art, this savings does not come at the expense of the search. When making a data base structure choice between the typical prior art tree structures, a trade-off must be made between inversion time and search time. If a particular tree structure is used to minimize the inversion time, then the search time will be relatively larger. With the present invention, this trade-off no longer need be made.

If a search needs to include the new data items, the search algorithm could either first merge the temporary SMTree with the main SMTree prior to the search, or the search could merely treat the search as two searches, one of the main SMTree and one of the temporary SMTree. This latter approach is likely to be the more efficient of the two. If the temporary SMTree is merged with the main SMTree prior to each search, there would be more accesses to secondary storage than for the search alone and some of the advantages of the present invention would not be realized to the fullest. If the temporary SMTree is maintained in primary storage for searches until it outgrows primary storage, no secondary storage accesses are needed prior to each search, reducing the search time.

Physically, the present invention stores a horizontal series of single-unit alternate lists in a single node. However, during the adding process, all units are treated logically as separate nodes. This greatly simplifies the process of adding a data item to an SMTree that has part of a node in common. For example, suppose the data item "justin" is to be added to the SMTree of Fig. 5. If a straight comparison of the unit 'j' to each node of the alternate list is made, 'j' does not exist because it is actually combined with the unit 'o' in the "jo" node. This means that in the adding process, the "jo" node has to be broken into a "j" node followed by an "o" node prior to adding the "ustin" node in the same alternate list as the "o" node. The implementation of the present invention logically treats all units as individual nodes so that comparisons are strictly unit-to-unit comparisons, rather than comparisons of combinations of units and multiple unit nodes.

The basic inversion mechanism traverses the SMTree searching for the data item that is being added, and when it reaches the point in the SMTree where a unit of the existing data item differs from the new data item, a new node

containing the different unit and all subsequent units is added to the alternate list following the last common unit. The new node is inserted into the SMTree following the last node of the subtree of which the first existing different unit is the root node.

Figs. 15 and 16 show the steps for adding the data item "justin" and identifier "84" to the SMTree of Fig. 9. The SMTree is traversed, looking for 'j' in the root alternate list, which is found in the node "jo" 212. Since the second unit of this node 212 is not 'u', the "jo" node 212 must be split into a "j" node 240 and an "o" node 242, as shown in Fig. 15. The new "j" node 240 keeps its position in the SMTree because it is still part of the root alternate list. The new "o" node 242 immediately follows the "j" node 240 because it is the first unit of the alternate list pointed to by the "j" node 240. Since it is the only member of its alternate list, the EAL flag 244 is set to "1". Next, a new "ustin" node 246 is created from the remainder of the new data item. The "ustin" node 246 is inserted into the SMTree following the last node of the subtree following the "o" node 242, which is the "ining" node 216 and its identifier 218, as in Fig. 16. Note that the "o" node EAL flag 244 has been reset to "0" because it is no longer the last node of the alternate list, and that the "ustin" EAL flag 248 has been set to "1" because it is now the last node of the alternate list. Then the "84" identifier 250 is inserted after the new "ustin" node 246.

The mechanics of inversion of the present invention are much more complex than this simple example, primarily because of the physical constraints under which the SMTree operates. In any practical implementation, size aspects of the SMTree have limitations. A node has a maximum size, limited by the amount of space allocated to the SIZ value 68. For example, if the node header 52 is allocated an eight-bit byte, the EOI flag 64, EAL flag 66, and NBP flag 70 each consume one bit,

leaving five bits for the SIZ value, which means a maximum node size of 32 units. Secondly, the block size is typically the size of a disk record, 512, 1024, or 2048 bytes. The consequence of these size limitations is that any data item add function must include not only routines for manipulating nodes, but also routines for manipulating blocks and subtrees. Suppose, for example, that a block has two subtrees and that the addition of a data item increases one subtree so that it no longer fits in the current block. One of the subtrees must be moved to another block. Either enough space must be found in an existing block for the moving subtree or a new block must be created. The block header 102 includes a block length value (BKL) 108 that indicates the total length of all of the subtrees in the block, and can be used to determine how much space is available in the block for another subtree or to expand an already resident subtree. After determining where the subtree will be moved to, the subtree is moved, the NBP pointer of the higher-level subtree that points to the moving subtree is changed to the new block address, and the new block header is changed to account for the moving subtree. Finally, the new data item is added.

In a more complicated scenario, suppose that a block has only one subtree and that the addition of a data item increases the subtree so that it no longer fits in the block. First, a new subtree must be carved from the now-oversized subtree, including creating an NBP pointer in the remaining subtree, compressing the remaining subtree, and creating a subtree header for the new subtree. Second, either enough space must be found in an existing block for the new subtree or a new block must be created, and the newly-created NBP pointer must be filled in with the block address. Next, the new subtree must be moved to the new block, including modifying the block header to account for the new subtree. And finally, the new data item is added.

Example pseudocode for adding a data item to the SMTree follows. The last routine, AddDataItem, is the routine that would be called by an external program to add the data item.

---

```
// AddSubtreeToBlock
// Add the given mtSubtree to the given mtBlock at the
// location specified by atIndex.  Space is inserted in the
// block to accommodate the new subtree.  Inserting space
// will of necessity move existing subtrees from atIndex
// forward.
AddSubtreeToBlock(mtBlock block,mtSubtree subtree,
                   integer atIndex)
    // Insert space in the block, moving the subtree at
    // atIndex and all "higher" subtrees to accommodate the
    // new subtree.  This routine should make space for all
    // the stored info for the subtree object, such as the
    // PRU and STL values, in addition to changing the
    // length of the block.
    INSERT_SUBTREE_SPACE(block,atIndex,subtree.STL)
    block.S_atIndex = subtree
end AddSubtreeToBlock
```

---

```
// CreateNewSubtreeInBlock
// Creates a new empty mtSubtree in the specified block.
// The atIndex integer specifies where in the list of
// subtrees to place the new subtree.  Effectively moves the
// current subtree at atIndex out of the way to make room
// for the new one.
CreateNewSubtreeInBlock(mtBlock block,integer atIndex,
                         mtUnit somePrevUnit)
    // Make a new empty subtree and set its PRU to that
    // passed in somePrevUnit.
    declare mtSubtree subtree
    subtree.PRU = somePrevUnit
    // Since it is new, the length is zero.
    subtree.STL = 0
```

```
    // Add the new subtree to the block.
    AddSubtreeToBlock(block,atIndex,subtree)
end CreateNewSubtreeInBlock
```

---

```
// CreateNewNode
// Make a new unattached node using the mtUnit data item
// passed.  This routine assumes j <= kMaxNodeLen.
CreateNewNode(mtUnit M1...Mj)
declare mtNode N
    i = 1
    loop while (i <= j)
        // Start at N.C0.
        N.Ci-1 = Mi
        i = i + 1
    end loop
    // SIZ is one less than j because a SIZ of zero implies
    // one unit in the node.
    N.Hdr.SIZ = j - 1
    // By itself, any node is an alternate list terminal and
    // a leaf.  These will need to be changed once the node
    // is inserted into a subtree.
    N.Hdr.EAL = TRUE
    N.Hdr.EOI = TRUE
    N.Hdr.NBP = FALSE
    N.BlkPtr = NULL
    return (N)
end CreateNewNode
```

---

```
// CreateNewNodeAtPosition
// Make a new node using the mtUnit data item passed.  If
// the data item is too long, several nodes will be created
// and chained together.  The node is placed at the current
// position, which is assumed to be the correct location in
// the subtree.
CreateNewNodeAtPosition(mtPosition pos,mtUnit M1...Mj,
                                 Boolean IsAltListTerminal)
    // If there is no current Node...
```

```
if (pos.Node = NULL)
    // The node is beyond the end of the SMTree.  In
    // this case, either the tree is empty so x will be
    // set to 1.  Or the node needs to be added to the
    // end of an existing list of nodes and x will be
    // set one past the end.
    x = GET_NODE_COUNT(pos.Sub) + 1
else
    // Get the index of the current node.
    x = GET_NODE_INDEX(pos.Node)
end if
// Save original node index for later.
y = x
i = 1
loop (while i <= j)
    // Set k to the minimum of the rest of the data item
    // length OR the maximum node length.
    k = MIN(j -(i - 1), kMaxNodeLen)
    declare mtNode node
    node = CreateNewNode(Mi...Mk)
    // Insert space in the subtree, moving the node at
    // atIndex and all "higher" nodes to accommodate the
    // new node.  It must make this space within the
    // block holding the subtree and move aside any
    // existing "higher" subtrees as well.  This routine
    // makes space for all the stored info for the node
    // object, such as the header and block pointer, in
    // addition to changing the subtree length.
    INSERT_NODE_SPACE(pos.Sub,x,node.Hdr.SIZ)
    pos.Sub.STL = pos.Sub.STL + SIZE_OF(node)
    // Add the node to the new space.
    pos.Sub.Nx = node
    if (k < j)
        // Still more nodes to come, so set the leaf
        // value of the node to FALSE.
```

```
                pos.Sub.Nx.Hdr.EOI = FALSE
                // More nodes need added so increment x
                x = x + 1
            end if
            // Set i to one past k in order to start out on
            // the units not added.  If i is beyond j then the
            // loop will terminate.
            i = k + 1
        end loop
        // Go back to the first added node.
        pos.Node = pos.Sub.Ny
        // Set the alternate list terminal nature of the first
        // node.
        pos.Node.Hdr.EAL = IsAltListTerminal
    end CreateNewNodeAtPosition
```

---

```
    // SplitNodeAtPosition
    // Divide the node into two nodes where the new node begins
    // with the unit pointed to by pos.
    SplitNodeAtPosition(mtPosition pos)
        // Make a block pointer to store the node's pointer
        // (if any).
        declare mtBlockPointer SavePointer
        if (pos.Node.Hdr.NBP = TRUE)
            SavePointer = pos.Node.BlkPtr
        else
            SavePointer = NULL
        end if
        declare mtNode SplitNode
        x = pos.u
        SplitNode = CreateNewNode(pos.Node.Cx...pos.Node.Ck)
        CollapseLen = pos.Node.Hdr.SIZ - pos.u + 1
        // Remove the "tail" units from the block by moving
        // higher nodes back.
        COLLAPSE_NODE_SPACE(pos,CollapseLen)
        pos.Sub.STL = pos.Sub.STL - CollapseLen
```

```
// Shrink the length of the original node.
pos.Node.Hdr.SIZ = pos.u - 1
// If the original node was a leaf node...
if (pos.Node.Hdr.EOI = TRUE)
    // It is impossible for this node to be a leaf now.
    pos.Node.Hdr.EOI = FALSE
    // The tail of the original node will now be a leaf.
    SplitNode.Hdr.EOI = TRUE
else
    // The original node was not a leaf so the split is
    // not either.
    SplitNode.Hdr.EOI = FALSE
end if
declare mtPosition PrevPos
if (SavePointer = NULL)
    // Get the index of the current node and increment.
    n = GET_NODE_INDEX(pos.Node)
    n = n + 1
else // The splitting node had a pointer.
    // Create a new block in the SMTree.
    declare mtBlock NewBlock
    NewBlock = CreateNewEmptyBlock(
                    GET_SMTREE_FOR_BLOCK(pos.Blk))
    // Set the pointer for the old node to point to the
    // new block.
    SetNodePointer(pos,POINTER(NewBlock))
    // Move the new node into the new block.
    CreateNewSubtreeInBlock(NewBlock,1,pos.Node.C0)
    PrevPos = pos
    pos = GetFirstBlockPosition(NewBlock)
    // Set the position stack to the correct chain
    // of blocks.
    pos.BlkStk = PrevPos.BlkStk
    declare mtBlockInfo NewBI
    NewBI.BlkPtr = POINTER(PrevPos.Blk)
```

```
        NewBI.s = GET_SUBTREE_INDEX(PrevPos.Sub)
        PUSH(pos.Blk,NewBI)
        n = 1
    end if
    // Insert space after the current node.  This must work
    // even if n is greater than the number of nodes in the
    // subtree.
    INSERT_NODE_SPACE(pos.Sub,n,SplitNode.Hdr.SIZ)
    // Add the complete length of the node to the subtree
    // header info.
    pos.Sub.STL = pos.Sub.STL + SIZE_OF(SplitNode)
    // Add the node to the new space.
    pos.Sub.N_n = SplitNode
    // Set the position to the split node.
    pos.Node = pos.Sub.N_n
    pos.u = 0
    // If there was a pointer in the original node...
    if (SavePointer <> NULL)
        // This is a complex procedure whereby the block
        // for the new node must be written out.  First
        // the node that SavePointer references must be
        // updated to point back to the correct unit in
        // the new block.  Next the new block must be
        // written.  Then finally, the original block
        // which held the original node must be written
        // to storage.  Things must be done in this order
        // because the original block may have to cascade
        // data into the new block.  This will not work
        // unless the new block exists in storage.
        // Set the pointer on the new split node.
        SetNodePointer(pos,SavePointer)
        declare mtPosition NextBlockPos
        NextBlockPos = pos
        NextBlockPos.u = NextBlockPos.Node.Hdr.SIZ
        // Go to the block pointed to by the original node
```

```
            // and set the previous unit of the subtree to
            // point back to the unit of the new node.  This
            // assures correct backtracking.
            // through the SMTree
            GoNextAlternateList(NextBlockPos)
            NextBlockPos.Sub.PRU = pos.Node.C0
            // Write out this "next" block to preserve the
            // change.  A direct write can be called since no
            // additional data has been added.
            WRITE_BLOCK(GET_SMTREE_FOR_BLOCK(NextBlockPos.Blk),
                        NextBlockPos.Blk)
            // Write out the new block.  A direct write can be
            // called since the block contains only one node
            // of data and therefore cannot be overflowed.
            WRITE_BLOCK(GET_SMTREE_FOR_BLOCK(pos.Blk),pos.Blk)
            // Read in the block previous to the original block
            // referred to here as RemoteBlock.  This is needed
            // to pass into the WriteBlockToStorage routine.
            declare mtBlockInfo RemoteBlockInfo
            RemoteBlockInfo = TOP(PrevPos.BlkStk)
            declare mtBlock RemoteBlock
            RemoteBlock = READ_BLOCK(
                          GET_SMTREE_FOR_BLOCK(PrevPos.Blk),
                          RemoteBlockInfo.BlkPtr)
            // Now call WriteBlockToStorage on the original
            // block.
            WriteBlockToStorage(PrevPos.Blk,RemoteBlock)
        end if
        return (pos)
    end SplitNodeAtPosition
```

---

```
    // CreateNewNodeBeforeAlternate
    // Insert a data item of mtUnits as a node into the current
    // mtPosition.  If necessary, split the current node.
    // Effectively inserts a new node with the qualification
    // that its first unit is the preceding alternate to the
```

```
// current alternate.
CreateNewNodeBeforeAlternate(mtPosition pos,mtUnit M1...Mj)
    // If the node needs to be added to a single unit
    // alternate list of the current node...
    if (pos.u > 0)
        // Split the current node at this point into two.
        SplitNodeAtPosition(pos)
    end if
    // Add the new node at the position.
    CreateNewNodeAtPosition(pos,M1...Mj,FALSE)
end CreateNewNodeBeforeAlternate
```

---

```
// CreateNewNodeAfterAlternate
// Make a new node(s) containing the data item of mtUnits
// M1...Mj as an alternate node following the current
// alternate.
CreateNewNodeAfterAlternate(mtPosition pos,mtUnit M1...Mj)
    // If the node needs to be added to a single unit
    // alternate list of the current node...
    if (pos.u > 0)
        // Split the current node at this point into two.
        SplitNodeAtPosition(pos)
    end if
    // If there is not a next alternate unit...
    if (GoNextAlternateUnit(pos) = FALSE)
        declare mtPosition LastPos
        // Save the last position.
        LastPos = pos
        // Find the last node derived from the current
        // alternate unit.
        ScanToLastNodeFromPosition(pos)
        // Make the last position an alternate list
        // non-terminal alternate.
        LastPos.Node.Hdr.EAL = FALSE
        // If the index of this node is the last node in the
        // subtree...
```

```
        if (GET_NODE_INDEX(pos.Node) =
                            GET_NODE_COUNT(pos.Sub))
            // Add the new node onto the very end of the
            // subtree by setting the position mtNode to
            // NULL.
            pos.Node = NULL
        else // There are more nodes in the subtree.
            // Go to the next node, which will need to be
            // moved aside for the new node.
            GoNextNode(pos)
        end if
        // Insert a new node at the correct position as a
        // terminal alternate.
        CreateNewNodeAtPosition(pos,M1...Mj,TRUE)
    else
        // Insert a new node at the correct position as a
        // non-terminal alternate.
        CreateNewNodeAtPosition(pos,M1...Mj,FALSE)
    end if
end CreateNewNodeAfterAlternate
```

---

```
// AddToNode
// Add units to the end of an existing mtNode.
AddToNode (mtPosition pos,mtUnit M1...Mj)
    // Make the new number of units added equal to the
    // minimum of the mtUnit data item minus the current
    // length OR the maximum length of a node.
    NodeLength = GET_UNIT_INDEX(pos.Node)
    AddLength = MIN(j - NodeLength,kMaxNodeLen - NodeLength)
    nextpos = pos
    // If there is a next node, space must be cleared in the
    // block.
    if (GoNextNode(nextpos) = TRUE)
        // Insert space in the subtree.
        x = GET_NODE_INDEX(nextpos.Node)
        INSERT_NODE_SPACE(nextpos.Sub,x,AddLength)
```

```
    end if
    // Set the new length of the node.
    pos.Node.Hdr.SIZ = pos.Node.Hdr.SIZ + AddLength
    i = NodeLength + 1
    n = 1
    // Fill the new cleared space with the new units.
    loop while (n <= AddLength)
        pos.Node.C_i = M_n
        i = i + 1
        n = n + 1
    end loop
    // Check if there are still more units to add.
    if (n < j)
        n = n + 1
        // In this case, the further units will be in a
        // continuation node(s), so this cannot be a leaf
        // node.
        pos.Node.Hdr.EOI = FALSE
        // Test going to the next node.
        if (GoNextNode(pos) = FALSE)
            // If there is no next node then pos is on the
            // last node of the subtree, so set it to NULL,
            // forcing CreateNewNodeAtPosition to add nodes
            // to the end of the subtree.
            pos.Node = NULL
        end if
        // Add any leftover units to a continuation node(s).
        CreateNewNodeAtPosition(pos,M_n...M_j)
    end if
end AddToNode
```

---

```
// CreateNewEmptyBlock
// Read a new block from the free store and set its length
// to zero.
CreateNewEmptyBlock(SMTree TheTree)
    declare mtBlock B
```

```
    READ_FREE_BLOCK(TheTree,B)
    B.BKL = 0
    return(B)
end CreateNewEmptyBlock
```

---

```
// CalculateAlternateSendoutSize
// Return the combined size of all the nodes off the current
// alternate item.  Not including the current node unless it
// is a leaf.
CalculateAlternateSendoutSize(mtPosition pos)
    // Set i to the first node index.
    i = GET_NODE_INDEX(pos)
    // If the node is not a leaf then skip the first node.
    if (pos.Node.Hdr.EOI = FALSE)
        i = i + 1
    end if
    // Find the last node off of the alternate item.
    ScanToLastNodeFromPosition(pos)
    // pos was updated to the very last node so get its
    // index.
    LastNodeIx = GET_NODE_INDEX(pos.Node)
    TotalSize = 0
    loop while (i <= LastNodeIx)
        // Add up the size of each node.
        TotalSize = TotalSize + SIZE_OF(pos.Sub.Ni)
        i = i + 1
    end loop
    return (TotalSize)
end CalculateAlternateSendoutSize
```

---

```
// SetNodePointer
// Add or change the current pointer referenced by the
// current node of pos.
SetNodePointer(mtPosition pos,mtBlockPointer BP)
    // If there is no pointer currently.
    if (pos.Node.Hdr.NBP = FALSE)
        // Add the next block pointer to the current node.
```

```
        pos.Node.Hdr.NBP = TRUE
        INSERT_POINTER_SPACE(pos)
        Block.BKL = Block.BKL + SIZE_OF(pos.Node.NBP)
        Pos.Sub.STL = Pos.Sub.STL + SIZE_OF(pos.Node.NBP)
    end if
    // Change the pointer.
    pos.Node.BlkPtr = BP
end SetNodePointer
```

---

```
// InsertNewSubtree
// Add a new subtree into an existing block, placing it
// in the correct position according to the sort order of
// the previous block.
InsertNewSubtree(mtBlock UpdateBlock,mtSubtree NewSubtree)
    // Scan through the update block to find the correct
    // insertion point for the new subtree.
    i = 1
    // Use the PRU value to determine the correct
    // location of the subtree in the block.  These must be
    // in sorted order to ensure correct handling of forward
    // searching in the SMTree.
    loop while (i <= GET_NUM_SUBTREES(UpdateBlock) AND
            UpdateBlock.Si.PRU < NewSubtree.PRU)
        i = i + 1
    end loop
    // Add the new subtree to the next block in the
    // correct sorted position.
    AddSubtreeToBlock(UpdateBlock,NewSubtree,i)
    return (i)
end InsertNewSubtree
```

---

```
// SendOutSubtrees
// Clip off parts of a subtree in Block after the base
// alternate list.  Find other blocks where these can go,
// creating blocks if necessary.  If blocks already exist
// that are subsequent to this one, they will be used in
// sorted order.  At the end of the routine, the changed
```

```
// blocks are written out.
SendOutSubtrees(mtBlock Block)
    // Make an array for storing blocks that will store the
    // new subtrees.  These blocks will be written out at
    // the end of the routine.
    declare BlockInfoArray NextBlockList
    // Keep sending out subtrees until the block is small
    // enough.
    loop while (Block.BKL > kMaxBlkLen)
        // Make a position object pointing into the block.
        declare mtPosition pos
        pos = GetFirstBlockPosition(Block)
        // This will mark the send out position.
        declare mtPosition SendOutPos
        SendOutPos = pos
        LargestSize = 0
        CurrentPointer = NULL
        LastPointer = NULL
        // Search through the entire root alternate list
        // for the largest subtree to send out.
        loop
            // If this node does not already have a
            // pointer...
            if (pos.Node.Hdr.NBP = FALSE)
                // Calculate the size of the subtree off
                // this alternate item.
                AlternateSize =
                        CalculateAlternateSendoutSize(pos)
                // If the subtree is larger than the
                // currently regarded largest then reset
                // the sending position.
                if (AlternateSize > LargestSize)
                    LargestSize = AlternateSize
                    SendOutPos = pos
                    // Set LastPointer to the pointer value
```

```
                // of the last alternate with a pointer
                // before this one.
                LastPointer = CurrentPointer
            end if
        else // The node has a pointer.
            CurrentPointer = pos.Node.BlkPtr
            if (LastPointer = NULL)
                // This ensures LastPointer will be
                // set if there were no pointers before
                // the final chosen alternate.  In this
                // case LastPointer will be the first
                // pointer after the chosen alternate.
                LastPointer = CurrentPointer
            end if
        end if
    until (GoNextAlternateUnit(pos) = FALSE)
    // Mark the current location of the send out
    // position.
    pos = SendOutPos
    // If the current node is the only one off this
    // alternate unit, then it must be clipped.
    if (SendOutPos.Node.Hdr.EOI = TRUE)
        // Clip the node pointed to by SendOutPos.
        // This causes the next alternate list to be
        // orphaned into its own node, suitable for
        // sending off to the new block.
        GoNextAlternateList(SendOutPos)
        SplitNodeAtPosition(SendOutPos)
    else // There are multiple nodes off this one.
        // In this case, move to the next node to be in
        // a position to send them all out to a new
        // block.
        x = GET_NODE_INDEX(SendOutNode.Node)
        x = x + 1
        SendOutPos.Node = SendOutPos.Sub.Nx
```

```
end if
declare mtBlock NextBlock
// Get the Block in which to insert the subtree.
if (LastPointer = NULL)
    // Create a new block.
    NextBlock = CreateNewEmptyBlock(
                            GET_SMTREE_FOR_BLOCK(Block))
    // Add this block's info to the Block info
    // array.
    declare BlockInfoItem NextBlockInfo
    NextBlockInfo.Ptr = POINTER(NextBlock)
    LastPointer = NextBlockInfo.Ptr
    NextBlockInfo.Block = NextBlock
    // Add the BlockInfoItem to the BlockInfoArray.
    ADD_TO_ARRAY(NextBlockList,NextBlockInfo)
else
    // Search through the block info array to see
    // if this block has already been used.  Since
    // the next blocks are not written until the
    // very end of the routine, we must continue
    // to use them instead of reading them again.
    x = GET_ARRAY_SIZE(NextBlockList)
    i = 1
    Found = FALSE
    loop while (i <= x AND Found = FALSE)
        if LastPointer = NextBlockList.Bi.Ptr
            NextBlock = NextBlockList.Bi.Block
            Found = TRUE
        end if
        i = i + 1
    end loop
    // If no block was found in the array...
    if (Found = FALSE)
        // Read the block from storage.
        NextBlock = READ_BLOCK(
```

```
                                GET_SMTREE_FOR_BLOCK(Block),
                                LastPointer)
                // Put its info into the array.
                declare BlockInfoItem NextBlockInfo
                NextBlockInfo.Ptr = LastPointer
                NextBlockInfo.Block = NextBlock
                // Add the BlockInfoItem to the
                // BlockInfoArray.
                ADD_TO_ARRAY(NextBlockList,NextBlockInfo)
            end if
        end if
        // Create the subtree with the nodes.
        declare mtSubtree SendOutSubtree
        // The previous unit for the subtree will be the
        // current alternate unit.
        SendOutSubtree.prevUnit = pos.Node.C0
        // Get the range of nodes which need to be sent.
        StartNodeIx = GET_NODE_INDEX(SendOutPos.Node)
        CurrNodeIx = StartNodeIx
        // Find the last node to be sent out.
        declare mtPosition lastNodePos
        lastNodePos = pos
        ScanToLastNodeFromPosition(lastNodePos)
        // pos was updated to the very last node so get its
        // index.
        LastNodeIx = GET_NODE_INDEX(lastNodePos.Node)
        i = 1
        TotalNodeLen = 0
        // Fill the new subtree with the nodes.
        loop while (CurrNodeIx <= LastNodeIx)
            SendOutSubtree.Ni = SendOutPos.Sub.NCurrNodeIx
            TotalNodeLen = TotalNodeLen +
                                SIZE_OF(SendOutSubtree.Ni)
            CurrNodeIx = CurrNodeIx + 1
            i = i + 1
```

```
        end loop
        InsertNewSubtree(NextBlock,SendOutSubtree)
        SetNodePointer(pos,LastPointer)
        // Remove the nodes from the current block entirely.
        pos.Node = pos.Sub.N_StartNodeIx
        COLLAPSE_NODE_SPACE(pos,TotalNodeLen)
        // Set the new length for the current block and
        // subtree.
        Block.BKL = Block.BKL - TotalNodeLen
        Pos.Sub.STL = Pos.Sub.STL - TotalNodeLen
    end loop // Main loop end.
    // Now that the block is small enough, it can be
    // written.
    WRITE_BLOCK(GET_SMTREE_FOR_BLOCK(Block),Block)
    // Now go through the changed next blocks and write
    // them.  This must be done here because
    // WriteBlockToStorage could change Block, which must
    // not be tampered with until all changes are made in
    // this routine.
    x = GET_ARRAY_SIZE(NextBlockList)
    i = 1
    loop while (i < x)
        // Write each changed next block to storage,
        // sending the current block in as the "previous"
        // block.  This can cause a cascading effect if the
        // next block is too large.
        WriteBlockToStorage(NextBlockList.B_i.Block,
                                      POINTER(Block))
        i = i + 1
    end loop
    return (Block)
end SendOutSubtrees
```

---

```
// ResetSubtreePointer
// Change the block pointer of a Subtree node to point to
// the block toBlock.
```

```
ResetSubtreePointer(mtBlock resetBlock, mtBlock toBlock,
                              mtUnit PRU)
     // Make a position object pointing into the reset block.
     declare mtPosition pos
     pos = GetFirstBlockPosition(resetBlock)
     // Find the alternate unit matching PRU.  This will be
     // the node that needs changed.
     if (FindAlternateUnit(PRU,pos))
          // Set the pointer to the new block.
          pos.Node.NBP = POINTER(toBlock)
     end if
     return (resetBlock)
end ResetSubtreePointer
```

---

```
// SplitBlock
// Go through the subtrees of a block and place them into a
// new block until it is split in half.  This routine also
// reads in the previous block and resets its pointers to
// point to the new block.
SplitBlock(mtBlock Block,mtBlock PrevBlock)
     // Get the size of half the block length.
     HalfSize = Block.BKL / 2
     // Create a new empty block.
     declare mtBlock NewBlock
     NewBlock =
               CreateNewEmptyBlock(GET_SMTREE_FOR_BLOCK(Block))
     // Now loop through the block sending out subtrees until
     // the block size is about half its previous size.
     TotalSize = 0
     i = 1
     loop while ((GET_SUBTREE_COUNT(Block) > 1) AND
                    (TotalSize < HalfSize))
          // Keep track of the total size of the new block.
          TotalSize = TotalSize + Block.S1.STL
          // Take the first subtree and append it into the new
          // block (the first subtree in Block will keep
```

```
            // changing because of this loop).
            AddSubtreeToBlock(newBlock,i,Block.S1)
            // Reset a pointer from the previous block to go to
            // the new block (i.e., the previous pointer used to
            // point to the current block but now the subtree it
            // identified has moved to the new block).
            ResetSubtreePointer(PrevBlock,newBlock,
                                    newBlock.S1.PrevUnit)
            // Remove the first subtree from Block.
            COLLAPSE_SUBTREE_SPACE(Block,1, Block.S1.STL)
            // Change the length of Block.
            Block.BKL = Block.BKL - Block.S1.STL
            i = i + 1
        end loop
        // Write the contents of the previous block.  No need to
        // check size here because data was changed but not
        // increased in size.
        WRITE_BLOCK(GET_SMTREE_FOR_BLOCK(PrevBlock),PrevBlock)
        // Write both the new and old blocks to storage.  Use
        // WriteBlockToStorage in case either or both is still
        // too large.
        WriteBlockToStorage(NewBlock,PrevBlock)
        WriteBlockToStorage(Block,PrevBlock)
    end SplitBlock
```

---

```
    // BreakUpBlock
    // Move information out of an overfull block to other
    // blocks.  If there is only one subtree in the block, then
    // that subtree must be portioned out so call
    // SendOutSubtrees().  If there is more than one subtree,
    // then simply send entire subtrees to other block with
    // SplitBlock().
    BreakUpBlock(mtBlock Block,mtBlock PrevBlock)
        // If there is only one subtree in the block...
        if (GET_SUBTREE_COUNT(Block) = 1)
            // Portions of the one subtree must be sent out as
```

```
        // new subtrees into other blocks.
        SendOutSubtrees(Block)
    else // There is more than one subtree in the block.
        // Split the block into two separate blocks.
        SplitBlock(Block,PrevBlock)
    end if
end BreakUpBlock
```

---

```
// WriteBlockToStorage
// Check the size of the block against the maximum block
// size (Note: this block should be a special editable block
// in primary memory that can actually grow larger than
// kMaxBlkLen).  If the block is greater than kMaxBlkLen,
// then it must be split.  Otherwise it can be written to
// storage.
WriteBlockToStorage(mtBlock Block,mtBlock PrevBlock)
    i = 1
    BlockLength = 0
    // Loop through all the subtrees and add up their sizes.
    // The BKL value of the block cannot be trusted here.
    // This is actually the routine that sets its value.
    loop (while i < GET_SUBTREE_COUNT(Block))
        BlockLength = BlockLength + Block.S_i.STL
    end loop
    // Set the current length of the block.
    Block.BKL = BlockLength
    // If the block is too large.
    if (BlockLength > kMaxBlkLen)
        // Break up the block until it is small enough.
        BreakUpBlock(Block,PrevBlock)
    else
        // Write the block to the current storage.
        WRITE_BLOCK(GET_SMTREE_FOR_BLOCK(Block),Block)
    end if
end WriteBlockToStorage
```

---

```
// AddDataItem
```

```
// Add a data item of mtUnits to an SMTree.  This procedure
// traces through the SMTree for as much of the mtUnit data
// item which already exists and then adds the rest of the
// data item to the SMTree as a new node(s).
AddDataItem(SMTree SMT,mtUnit M1...Mj)
    // Go to the very first unit of the SMTree.
    declare mtPosition pos
    pos = GetFirstPosition(SMT)
    i = 1
    // If there are no nodes then the tree is completely
    // empty.  Simply add the first subtree and node.
    if (pos.Node = NULL)
        // Create a new empty subtree at the first position
        // in the block.
        pos.Sub = CreateNewSubtreeInBlock(pos.Blk,1,NULL)
        // Create a new node in the new subtree from the
        // data item.
        CreateNewNodeAtPosition(pos,M1...Mj)
    else // The tree is not empty.
        // As long as alternates exist for M, move forward
        // in the tree.
        loop while (i <= j
                        AND FindAlternateUnit(Mi,pos) = TRUE
                        AND GoNextAlternateList(pos) = TRUE)
            i = i + 1
        end loop
        // If there are still units left in M then add what
        // is left to the tree, starting within the last
        // checked alternate list.
        if (i <= j)
            x = pos.u
            // If the last checked alternate list contains
            // the unit Mi...
            if (pos.Node.Cx = Mi)
                // The search ended on one of the units and
```

```
                    // there was no next alternate list.  In this
                    // case we must be at the very end of a node,
                    // so simply tack on the remaining units.
                    i = i + 1
                    // Check i against j again in case it was on
                    // last item.
                    if (i <= j)
                        AddToNode(pos,Mi...Mj)
                    end if
                else // The find failed on the current alternate
                    // list.  Add a new node(s) to the current
                    // alternate list with the remaining units.
                    if (pos.Node.Cx < Mi)
                        // Add the new information as a previous
                        // alternate node.
                        CreateNewNodeBeforeAlternate(pos,M1...Mj)
                    else
                        // Add the new information as a next
                        // alternate node.
                        CreateNewNodeAfterAlternate(pos,M1...Mj)
                    end if
                end if
            end if
        end if
        // Get the previous block from the block stack.  Needed
        // in case WriteBlockToStorage must break up the block
        // because it is too large.
        declare mtBlockInfo bkInfo
        bkInfo = POP(pos.BlkStk)
        declare mtBlock PrevBlock
        PrevBlock = READ_BLOCK(SMT,bkInfo.BlkPtr)
        // Write the block to storage.
        WriteBlockToStorage(pos.Blk,PrevBlock)
    end AddDataItem
```

5. Merging SMTrees the Horizontal Embodiment

As discussed above, it is occasionally necessary to merge two SMTrees. Fig. 17 shows a temporary SMTree, consisting of the data items "carl, "carol", "juan", and "julia", that is to be merged in to the SMTree of Fig. 5. Figs. 18 and 19 diagram the merging process. Essentially, the SMTree merging process is the process of merging the root alternate list 274 of the temporary SMTree 270 with the root alternate list 272 of the SMTree 50 that the temporary SMTree is to merged into, the main SMTree.

As with the process of adding a data item to an SMTree, all units are treated logically as separate nodes. This greatly simplifies the process of merging SMTrees and is assumed in the following description.

The main root alternate list 272 is traversed, looking for the first unit of the temporary root alternate list 274, the unit 'c'. Since 'c' is not found in the main root alternate list 272, the unit 'c' is added to the main alternate list 272, and all subsequent units are added to the main SMTree. This means that the nodes "car", "l", "ol", and their associated identifiers are inserted into the main SMTree, as at 276 in Fig. 18. Optionally, the added nodes will be removed from the temporary SMTree 270. One purpose of the removal is to ensure that there are no duplicate entries if, for example, a search interrupts the merging process. However, if the search is designed to terminate after finding the first occurrence of the data item, there is no need to remove the node from the temporary SMTree 270.

Next, the main root alternate list 272 is traversed, looking for the next unit (now the first unit after removal of the previous first unit 'c') of the temporary root alternate list 274, the unit 'j'. In general, each traverse looking for the next unit will start over at the beginning of the main root alternate list 272. However, since most SMTrees will be sorted in some kind of logical order, such as

alphabetically, it will typically not be necessary to start over; the search for the next unit can begin where the previous search ended.

The unit 'j' is found in the node "jo" 212. Since the second unit of the "jo" node is not 'u', the "jo" node must be split into a "j" node and an "o" node, as described above with reference to adding a data item and as at 278 in Fig. Then the rest of the temporary SMTree 270 following the 'u' unit is added to the main SMTree 50. A new 'u' unit is inserted into the alternate list that starts with the new 'o' unit, and the nodes "an" and "lia" and their associated identifiers are inserted after the "u" node, as at 280.

As with the addition of a data item to an SMTree, the mechanics of merging two SMTrees is much more complex than this simple example. All of the same issues with the physical constraints imposed by the computer and storage arise with even greater force than when a single data item is being added.

Recall the explanation above regarding how adding data items to a temporary SMTree and then merging it with the main SMTree substantially reduces inversion time. The structure of the SMTree lends itself well to further reductions in inversion time through the use of parallel processing methods. As indicated, a merge of two SMTrees is accomplished by merging the two root alternate lists of the SMTrees. A single processor traverses through each root alternate list from the first list member to the terminal member. If a second processor is used, each root alternate list could be split at the same point, for example, in the middle of the list. The split does require that the two parts of the root alternate list have no secondary storage in common, otherwise it is possible that the two processors would be working in the same memory block at odds with each other. Since, from a logical viewpoint, none of the first part of the split alternate list is common to the second part

of the same alternate list, each processor can work completely independently on its part of the alternate lists. Consequently, the inversion time can be reduced to as little as half the time as compared to a single processor. And, in general, as more processors are added, the inversion time can be reduced proportionately.

Example pseudocode for merging two SMTrees using one processor follows. The last routine, MergeSMTrees, is the routine that would be called by an external program to merge two SMTrees.

```
// IsInBlockRoot
// Return TRUE if the current node is in the root alternate
// list of the block. Note: This function does not change
// the mtPosition parameter, only the local copy.
IsInBlockRoot(mtPosition pos)
    // Set the position to the first item in the node.
    pos.u = 0
    // First, back up to the first mtUnit in the current
    // alternate list.
    loop while (GoPreviousAlternateUnit(pos))
    end loop
    // Once here, discover if this first alternate item
    // node is in the root alternate list of the block.
    x = GET_NODE_INDEX(pos.Node)
    // If the current mtNode is the first one in the
    // subtree...
    if (x = 1)
        // The mtPosition is in the root of the block.
        return (TRUE)
    else
        // The mtPosition is not in the root of the block
        return (FALSE)
    end if
end IsInBlockRoot
```

```
// PickBlock
// Check inside pos to find an existing next block and
// return it.  If no block exists, make a new one inside the
// SMTree referenced by pos and return it.
PickBlock(mtPosition pos)
    declare mtBlockPointer ReturnPointer
    ReturnPointer = NULL
    u = pos.u
    TestUnit = pos.Node.C_u
    // Loop to the top of the alternate list in pos.
    loop until (GoPreviousAlternateUnit(pos) = FALSE)
    // Loop through the entire alternate list, checking
    // for the closest pointer to the original location
    // referenced by pos.
    loop
        if (pos.Node.Hdr.NBP = TRUE)
            ReturnPointer = pos.Node.BlkPtr
        end if
    until (GoNextAlternateUnit(pos) = FALSE OR
            (ReturnPointer <> NULL AND
            TestUnit >= pos.Node.C_0))
    declare mtBlock ReturnBlock
    // If no pointer was found...
    if (ReturnPointer = NULL)
        // Make a new block for the tree.
        ReturnBlock = CreateNewEmptyBlock(
                        GET_SMTREE_FOR_BLOCK(pos.Blk))
    else // A suitable pointer was discovered.
        // Return the block referenced by the pointer.
        ReturnBlock =
                   READ_BLOCK(GET_SMTREE_FOR_BLOCK(pos.Blk),
                   ReturnPointer)
    end if
    return (ReturnBlock)
end PickBlock
```

```
// AppendAlternateTree
// Add an entire alternate list with all its sub alternate
// lists from FromPos into ToPos.  This is done by adding
// all the successive nodes from the alternate list pointed
// to by FromPos.  Assumes FromPos is on the first alternate
// unit in the alternate list.  DeleteOriginal, if TRUE,
// causes the added nodes to be removed from the SMTree
// referenced by FromPos.
AppendAlternateTree(mtPosition ToPos,mtPosition FromPos,
                              Boolean DeleteOriginal)
     declare mtPosition EndPos
     EndPos = FromPos
     // Go to the last alternate unit.
     loop until (GetNextAlternateUnit(EndPos) = FALSE)
     end loop
     // Go to the very last node from the last alternate.
     ScanToLastNodeFromPosition(EndPos)
     // Get the index numbers of the first and last node
     // to be appended.
     j = GET_NODE_INDEX(EndPos.Node)
     i = GET_NODE_INDEX(FromPos.Node)
     // Get the index of the append location if there is no
     // node in the current subtree (Assumes a subtree
     // exists).
     if (ToPos.Node = NULL)
         // The current node is zero (k will increment).
         k = 0
     else
         k = GET_NODE_INDEX(ToPos.Node)
     end if
     // Start one past the current node in ToPos
     k = k + 1
     TotalSize = 0
     declare mtPosition FirstFromPos
     FirstFromPos = FromPos
```

```
    // Loop through all the nodes in FromPos for its current
    // alternate list, adding them at the point just past
    // the node pointed to by ToPos.
    loop while (i <= j)
        declare mtNode node
        node = FromPos.Sub.Ni
        TotalSize = TotalSize + SIZE_OF(node)
        // Insert space in the subtree, moving the node
        // currently there and all "higher" nodes to
        // accommodate the new node.
        INSERT_NODE_SPACE(ToPos.Sub,k,node.Hdr.SIZ)
        ToPos.Sub.STL = ToPos.Sub.STL + SIZE_OF(node)
        // Add the node to the new space.
        pos.Sub.Nk = node
        k = k + 1
        i = i + 1
    end loop
    if (DeleteOriginal = TRUE)
        // Remove all the space in FromPos associated with
        // the nodes added to ToPos.
        COLLAPSE_NODE_SPACE(FirstFromPos,TotalSize)
        // Set the lengths of the subtree and block to
        // reflect the removed nodes.
        FirstFromPos.Sub.STL = FirstFromPos.Sub.STL -
                                          TotalSize
        FirstFromPos.Blk.BKL = FirstFromPos.Blk.BKL -
                                          TotalSize
    end if
end AppendAlternateTree
```

---

```
// TransplantBlocksFromSubtree
// Using a pointer in ToPos, which points to a block of
// another tree, copy all the blocks in the FromTree to
// new blocks in the tree pointed to by ToPos.  This
// procedure is used during a merge to dump the unique
// contents of the FromTree into new blocks of the tree
```

```
// referenced by ToPos.  This is a recursive routine.
TransplantBlocksFromSubtree(mtPosition ToPos,
                                        SMTree FromTree)
    // Used to keep track of the last block used in case
    // it needs to be used again while going down the
    // subtree list.
    declare mtBlockPointer FormerBlockPtr
    FormerBlockPtr = NULL
    // Loop through all the alternate units in the root
    // alternate list of the block, checking each one for
    // a pointer.
    loop
        // If ToPos node has a pointer and that pointer
        // is not the same as the former block pointer
        if (ToPos.Node.Hdr.NBP = TRUE AND
                ToPos.Node.BlkPtr <> FormerBlockPtr)
            // Set the former block pointer to this one
            // for checking in the next round.
            FormerBlockPtr = ToPos.Node.BlkPtr
            declare mtBlock FromBlock
            // Get the entire block pointed to by the
            // pointer in ToPos node.
            FromBlock =
                    READ_BLOCK(FromTree,ToPos.Node.BlkPtr)
            // Make a new block in which to dump the
            // contents of FromBlock.
            declare mtBlock NewBlock
            NewBlock = CreateNewEmptyBlock(
                          GET_SMTREE_FOR_BLOCK(ToPos.Blk))
            // Copy the entire contents of the block.
            // Naturally this copies the pointers as well,
            // hence the idea of using the pointer in ToPos
            // to get the next block as this routine
            // recurses.
            COPY_BLOCK(NewBlock,FromBlock)
```

```
                // Get the first position in the new block.
                declare mtPosition CopyPos
                CopyPos = GetFirstBlockPosition(NewBlock)
                i = 1
                // Loop through all the subtrees in the new
                // block, recursively calling this routine in
                // order to create new blocks for any pointers
                // that exist in their root alternate lists.
                loop while (i <= GET_NUM_SUBTREES(NewBlock))
                    CopyPos.Sub = CopyPos.Blk.Si
                    CopyPos.Node = CopyPos.Sub.N1
                    CopyPos.u = 0
                    // Recursion takes care of making new blocks
                    // for the pointers and changing the pointer
                    // values to reference those new blocks.
                    TransplantBlocksFromSubtree(CopyPos,FromTree)
                    i = i + 1
                end loop
                // Change the pointer to reference the new
                // block.
                SetNodePointer(ToPos,POINTER(NewBlock))
                // Write the new block to storage.  Assume the
                // original block referenced by ToPos will be
                // written outside of this routine.
                WriteBlockToStorage(NewBlock,ToPos.Blk)
            end if
        // Keep moving down the alternate list of ToPos.
        until (GoNextAlternateUnit(ToPos) = FALSE)
end TransplantBlocksFromSubtree
```

---

```
// SpreadOutBlock
// Take the alternate list of the current position and move
// it into a new block as the root alternate list.  To
// accomplish this, all previous multi-item alternate
// lists must be moved to their own blocks.  This procedure
// will be called recursively for each alternate list behind
```

```
// the original request.
SpreadOutBlock(mtPosition pos,mtBlock PrevBlock)
    // Check if it is already in the root of the block.
    // If not then begin the process.  This will end the
    // recursion as well as check the initial settings.
    if (IsInBlockRoot(pos) = FALSE)
        // A string of mtUnits.
        declare MT_UNIT_STRING FindString
        i = 0
        // Add into the string each alternate list behind
        // the current one until the root block is reached.
        loop
            j = pos.u
            // Get all the units from the node.
            // This loop saves time by not calling
            // GoPreviousAlternateList for each unit.
            loop
                i = i + 1
                FindString.Si = pos.Node.Cj
                j = j - 1
            until j < 0
            pos.u = 0
            // Go back to the previous alternate list.
            GoPreviousAlternateList(pos)
        until (IsInBlockRoot(pos) = TRUE)
        // Make a new block to hold the new root alternate
        // list.
        declare mtBlock NewBlock
        NewBlock = CreateNewEmptyBlock(
                                GET_SMTREE_FOR_BLOCK(pos.Blk))
        declare mtPosition NewBlockPos
        NewBlockPos = GetFirstBlockPosition(NewBlock)
        // Set the new block position stack to the correct
        // chain of blocks.
        NewBlockPos.BlkStk = pos.BlkStk
```

```
            declare mtBlockInfo NewBI
            NewBI.BlkPtr = POINTER(pos.Blk)
            NewBI.s = GET_SUBTREE_INDEX(pos.Sub)
            PUSH(NewBlockPos.BlkStk,NewBI)
            NewBlockPos.Sub = CreateNewSubtreeInBlock(NewBlock,
                                                     1,pos.Node.C0)
            // Advance to the next node in the original
            // position.
            declare mtPosition NextPos
            NextPos = pos
            GoNextNode(NextPos)
            // Append the alternate list with all its nodes
            // from the one block to the other, deleting the
            // original.
            AppendAlternateTree(NewBlockPos,NextPos,TRUE)
            SetNodePointer(pos,POINTER(NewBlock))
            // Reverse the string created earlier so it can be
            // used to find the original position in the new
            // block.
            REVERSE_STRING(FindString)
            NewBlockPos = GetFirstBlockPosition(NewBlock)
            FindFromPosition(NewBlockPos,FindString)
            // Write the original (now changed) block to
            // storage.
            WriteBlockToStorage(pos.Blk,PrevBlock)
            // Recursively call SpreadOutBlock to move the
            // next alternate list to a new block (if any).
            SpreadOutBlock(NewBlockPos,pos.blk)
            // Leave pos at the final destination in the last
            // created block.
            pos = NewBlockPos
        end if
        return (pos)
    end SpreadOutBlock
```

---

```
// AddAlternateTree
```

```
// Use two mtPosition objects.  Place the contents of
// FromPos into ToPos, starting at the next alternate list
// from each.  This procedure assumes that ToPos and FromPos
// are identical in their respective chains of alternates,
// up to and including the current alternate unit.
AddAlternateTree(mtPosition ToPos,mtPosition FromPos)
    FoundPointer = FALSE
    // Set a boolean to reflect if the next alternate list
    // will be in a new block.
    FromStartsAtPointer = FromPos.Node.Hdr.NBP
    // Test for if FromPos is on the last alternate list
    // (in which case do nothing).
    if (GoNextAlternateList(FromPos) = TRUE)
        // If the "from" list began on a pointer, then it
        // will now point in a new block.  This block must
        // be checked to see if it has pointers.
        if (FromStartsAtPointer = TRUE)
            // Use a copy of FromPos so FromPos will not
            // change.
            declare mtPosition CheckPos
            CheckPos = FromPos
            // Check every node in the root alternate list
            // of the block to see if a pointer exists.
            loop
                FoundPointer = CheckPos.Node.Hdr.NBP
            until (FoundPointer = TRUE OR
                    GoNextAlternateUnit(CheckPos) = FALSE)
        end if
        // If a pointer exists within FromPos...
        if (FoundPointer = TRUE)
            // Check that the ToPos is in the root of the
            // Block.  If not, then this alternate list must
            // be put in its own block first.
            if (IsInBlockRoot(ToPos) = FALSE)
                declare mtBlockInfo PrevBlockInfo
```

```
        // Read the top of the block stack without
        // popping.
        PrevBlockInfo = TOP(ToPos.BlkStk)
        declare mtBlock PrevBlock
        // Get the previous block for the next call.
        PrevBlock = READ_BLOCK(
                  GET_SMTREE_FOR_BLOCK(ToPos.Blk),
                  PrevBlockInfo.BlkPtr)
        // Spreading blocks is a process of adding
        // new blocks to the tree until the current
        // alternate list resides in its own block
        // as the root of that block.  This must be
        // done because the ToPos alternate list
        // is going to point to a new block (copied
        // from the other merging tree) and only
        // root alternate lists can have pointers.
        SpreadOutBlock(ToPos,PrevBlock)
    end if
    // Make a new block in which to copy the subtree
    // pointed to by FromPos.  Must first copy the
    // subtree, not the block, because the other
    // subtrees in the block are not part of this
    // alternate tree.
    declare mtBlock NextBlock
    // Find an existing block from the ToPos or make
    // a new one.
    NextBlock = PickBlock(ToPos)
    SubIndex =
           InsertNewSubtree(NextBlock,FromPos.Sub)
    // Set the node pointer at this position.  This
    // will make the pointer if one does not exist.
    SetNodePointer(ToPos,POINTER(NextBlock))
    declare mtPosition NextPos
    NextPos = GetFirstBlockPosition(NextBlock)
    // Set the subtree in NextPos to the one just
```

```
        // added.
        NextPos.Sub = NextPos.Blk.S_SubIndex
        NextPos.Node = NextPos.Sub.N_1
        NextPos.u = 0
        // Copy all the blocks in the FromPos SMTree
        // into the ToPos tree by making new blocks and
        // changing the pointers to reflect the new
        // locations.
        TransplantBlocksFromSubtree(NextPos,
                    GET_SMTREE_FOR_BLOCK(FromPos.Blk))
        // Must write the original block because
        // TransplantBlocksFromSubtree will not do so.
        WriteBlockToStorage(NewBlock,ToPos.Blk)
    else // There are no pointers in FromPos.
        // Append the FromPos alternate list with all
        // its nodes into the ToPos.  Pass FALSE to
        // indicate that the original nodes should not
        // be deleted.
        AppendAlternateTree(ToPos,FromPos,FALSE)
    end if
    // Finally the original block referenced by ToPos
    // must be written out to storage.
    declare mtBlockInfo bkInfo
    // Get the previous block from the stack in ToPos
    // so that it can be referenced by
    // WriteBlockToStorage if necessary.
    bkInfo = POP(ToPos.BlkStk)
    declare mtBlock PrevBlock
    // Get the previous block.
    PrevBlock =
            READ_BLOCK(GET_SMTREE_FOR_BLOCK(ToPos.Blk),
            bkInfo.BlkPtr)
    WriteBlockToStorage(ToPos.Blk,PrevBlock)
    // Push the previous block stack info item back onto
    // the ToPos stack.
```

```
        PUSH(ToPos.BlkStk,bkInfo)
    end if
end AddAlternateTree
```

---

```
// MergeAlternateLists
// Starting with two mtPosition objects, merge the contents
// of each current alternate list.  Loop through every unit
// in the alternate list pointed to by FromPos.  As FromPos
// moves down the list, test the matches between the "to"
// alternate list and the "from" alternate list.  For any
// alternate not existing in the "from" list, simply add the
// remaining portion of the "to" list.  Skip any alternates
// in the "to" list not existing in the "from" list.
// Recursively call this routine for any units that exist
// both in the "to" and "from" alternate lists.
MergeAlternateLists(mtPosition ToPos,mtPosition FromPos)
    // Get the first alternate unit in the "to" alternate
    // list.
    x = ToPos.u
    declare mtUnit ToAlt
    ToAlt = ToPos.Node.Cx
    Terminated = FALSE
    // Loop through every unit in the alternate list pointed
    // to by FromPos.
    loop
        // Get the current alternate unit in the "from"
        // alternate list.
        y = FromPos.u
        declare mtUnit FromAlt
        FromAlt = FromPos.Node.Cy
        // If the alternate in the merge "to" alternate list
        // is less than the alternate in the "from"
        // alternate list, loop through the "to" list until
        // either the "to" list terminates or the alternate
        // unit in the "to" list is greater than or equal to
        // the alternate unit in the "from" list.
```

```
if (Terminated = FALSE AND ToAlt < FromAlt)
    loop
        // Go to the next alternate unit in the "to"
        // list.
        if (GoNextAlternateUnit(ToPos) = FALSE)
            // Went past the end of the "to" list.
            // Terminate this loop.
            Terminated = TRUE
        else
            // Get the current alternate unit in
            // the "to" list.
            x = ToPos.Node.u
            ToAlt = ToPos.Node.Cx
        end if
    until (ToAlt >= FromAlt OR Terminated = TRUE)
end if
// If the next "to" unit is after the "from" unit
// OR there are no more units in the "to" list.
if (FromAlt = ToAlt)
    // In this case both units exist in the
    // alternate lists, so copy the positions and
    // move forward in the trees, recursively
    // calling this routine to check the next
    // alternate lists against each other.
    NextToPos = ToPos
    NextFromPos = FromPos
    // If there is a next "from" alternate list...
    // If this fails, then do nothing since there
    // is nothing to add.
    If (GoNextAlternateList(NextFromPos) = TRUE)
        // If there is a next "to" alternate list...
        If (GoNextAlternateList(NextToPos) = TRUE)
            // Call merge on both positions.
            MergeAlternateLists(NextToPos,
                                NextFromPos)
```

```
            else // No next "to" list exists.
                // Add the entire alternate tree in
                // FromPos onto the current ToPos
                // position.
                j = GET_UNIT_COUNT(FromPos.Node)
                if (y < j)
                    y = y + 1
                    // Add the portion of the FromPos
                    // node which does not exist in the
                    // ToPos alternate list.
                    AddToNode(ToPos,
                       FromPos.Node.Cy...FromPos.Node.Cj)
                    FromPos.u = j
                    // Advance ToPos to the end of node.
                    ToPos.u = GET_UNIT_COUNT(ToPos.Node)
                end if
                // Add the rest of the nodes in FromPos
                // up to the next alternate item.
                AddAlternateTree(ToPos,FromPos)
            end if
        end if
    else if (FromAlt < ToAlt)
        j = GET_UNIT_COUNT(FromPos.Node)
        // Make a new node in the alternate list out of
        // the first node in FromPos.
        CreateNewNodeBeforeAlternate(ToPos,
                          FromPos.Node.Cy...FromPos.Node.Cj)
        // Advance FromPos to end of the node.
        FromPos.u = j
        // Advance ToPos to the end of node.
        ToPos.u = GET_UNIT_COUNT(ToPos.Node)
        // Add the rest of the nodes in FromPos
        // up to the next alternate item.
        AddAlternateTree(ToPos,FromPos)
    else if(Terminated = TRUE)
```

```
            j = GET_UNIT_COUNT(FromPos.Node)
            // Add the alternate unit from the "from" list
            // plus the entire tree that stems from it.
            CreateNewNodeAfterAlternate(ToPos,
                                FromPos.Node.Cy...FromPos.Node.Cj)
            // Advance FromPos to end of the node.
            FromPos.u = j
            // Advance ToPos to the end of node.
            ToPos.u = GET_UNIT_COUNT(ToPos.Node)
            AddAlternateTree(ToPos,FromPos)
        end if
    until (GoNextAlternateUnit(FromPos) = FALSE)
end MergeAlternateLists
```

---

```
// MergeSMTrees
// Merge the SMTree "BTree" into the SMTree "ATree".
MergeSMTrees(SMTree ATree,SMTree BTree)
    // Use a position object for each tree.
    declare mtPosition posA
    declare mtPosition posB
    // Get the first (root) alternate of each tree.
    posA = GetFirstPosition(ATree)
    posB = GetFirstPosition(BTree)
    // Simply merge the two root alternate lists.  The
    // rest of the trees are taken care of automatically.
    MergeAlternateLists(posA,posB)
end MergeSMTrees
```

---

6. Removing Data Items from the Horizontal Embodiment

Occasionally during the life of a data base, data items must be removed. The basic mechanism for removing a data item traverses the SMTree searching for the leaf node of the data item to be removed, and removes nodes in reverse order up to and including the node of a multiple-unit alternate list.

Like the add and merge operations, the mechanics of removing a data item are made more complex by the physical constraints under which the SMTree operates. Because there are multiple blocks and hierarchies of subtrees, a data item to be removed may cross a subtree/block boundary. If this should occur, the lower-level subtree and the block pointer to that subtree in the higher-level subtree are removed. And if the removed subtree is the only one in a block, the block is removed or freed.

Optionally, a method of compacting nodes can be implemented. If, for example, the last alternate list of the removed data item has only two members, once the data item is removed, that alternate list only has one member. It would then be possible to combine that remaining node with the node that precedes it. For example, if the data item "joe" was removed from the tree of Fig. 8, the "jo" node and the "ining" node could be combined into a single "joining" node.

Example pseudocode for removing a data item from the SMTree follows. RemoveDataItem is the routine that would be called by an external program to remove the data item.

---

```
// RemoveDataItem
// Find the string of units in the SMTree. If it exists,
// remove all references to it by removing successive nodes
// from the end of the tree.  Clean up any empty blocks
// along the way.
RemoveDataItem(SMTree SMT,mtUnit S1...Sj)
    FirstBlockPtr = GET_ROOT_BLOCK_POINTER(SMT)
    // Go to the very first part of the SMTree.
    declare mtPosition pos
    // Check for the existence of the item first.  This
    // also puts pos at the end of the item.
    if (FindDataItem(pos,SMT,S1...Sj))
        Done = FALSE
        loop
            x = GET_NODE_INDEX(pos.Node)
```

```
declare mtPosition altPos
// If the current position is the last node in
// the alternate list then the previous node
// must become the new terminal node.
if (pos.Node.Hdr.EAL = TRUE)
    altPos = pos
    // If there is a previous alternate.
    if (GoPreviousAlternateUnit(altPos))
        // Change the previous node to a terminal
        // node.
        altPos.Node.Hdr.EAL = TRUE
        // If pos.Node is not the only alternate,
        // stop deleting because the previous
        // nodes are part of other items.
        Done = TRUE
    end if
else // This is not a terminal alternate.
    // In this case since there are more
    // alternate items, any previous lists
    // cannot be removed since the other
    // alternates depend on them.
    // The delete is finished.
    Done = TRUE
end if
altPos = pos
// Attempt to go to the previous alternate
// list.
HasPrev = GoPreviousAlternateList(pos)
ShrinkSize = SIZE_OF(altPos.Node)
COLLAPSE_NODE_SPACE(altPos, ShrinkSize)
altPos.Sub.STL = altPos.Sub.STL - ShrinkSize
altPos.Blk.BKL = altPos.Blk.BKL - ShrinkSize
if (altPos.Blk.BKL = 0)
    // The block is empty and must be deleted.
    EmptyBlockPtr = POINTER(altPos.Blk))
```

```
                // Check to make sure it is not the first
                // block in the tree, which must not be
                // removed.
                if (EmptyBlockPtr <> FirstBlockPtr)
                    FREE_BLOCK(EmptyBlockPtr)
                end if
                // If there is a previous alternate list.
                if (HasPrev)
                    // pos will point to the node that
                    // points to the now deleted block.
                    // Remove the pointer from this node.
                    pSize = SIZE_OF(pos.Node.NBP)
                    COLLAPSE_POINTER_SPACE(altPos,pSize)
                    pos.Sub.STL = pos.Sub.STL - pSize
                    pos.Blk.BKL = pos.Blk.BKL - pSize
                    // Set node pointer indicator to
                    // FALSE.
                    pos.Node.Hdr.NBP = FALSE
                    // Set node to a leaf node.
                    pos.Node.Hdr.EOI = TRUE
                end if
            end if
        until (Done = TRUE)
    end if
end RemoveDataItem
```

---

## Vertical Embodiment

In the vertical embodiment 400, units 402 are compressed vertically into nodes 404. In other words, each node 404 is an alternate list, as shown in Fig. 20 for the tree of Fig. 2. Instead of 38 separate nodes 34, each containing one unit 32, there are only 31 nodes 404, shown in solid boxes, each containing an alternate list 406. Note that there are many more nodes in the vertical embodiment, at least for this example, than there are for the horizontal embodiment (31 nodes versus 12 nodes).

In the vertical embodiment, the nodes 404 and identifiers 408 are stored sequentially in memory first from top to bottom then from left to right. That is, the nodes and identifiers are stored in the following order: [ajs], [bd], [o], [et], [b], [a], [ei], [m], [a], [i], [m], [56], [n], [e], [n], [e], [a], [i], [s], [dfl], [18], [n], [n], [t], [26], [o], [e], [t], [g], [e], [r], [y], [11], [38], [r], [d], [0], [77], [63].

Theoretically, an SMTree of the vertical embodiment is traversed in a manner similar to an SMTree of the horizontal embodiment. In practice, however, the implementation would be much more difficult. The amount of time it would take to find a particular data item is greater than with the horizontal embodiment. In the horizontal SMTree of Fig. 6, in order to find the data item "joining", six nodes and identifiers need to be read. On the other hand, in the vertical SMTree of Fig. 20, 29 nodes and identifiers must be read before finding the data item "joining", and another five nodes and identifiers must be read to reach the identifier for the data item "joining".

Because the horizontal embodiment is more efficient in most applications, it is preferred over the vertical embodiment. Consequently, the vertical embodiment has not been implemented. It is included here to show that the embodiment, although possible, is not a practical application of the present invention.

<u>Hybrid Embodiment</u>

It is possible to combine characteristics of the horizontal embodiment with characteristics of the vertical embodiment to result in a hybrid embodiment 500. In the hybrid embodiment 500, an alternate list 506 is compressed into a node 504 and a horizontal series of unique units 502 are compressed into a node 504, as in Fig. 21. Instead of 38 separate nodes 34, each containing one unit 32, there are

only 13 nodes 504, shown in solid boxes, each containing an alternate list 506 or a unique series 510.

There are two possible ways to store the hybrid embodiment 500 in memory, either horizontally or vertically. Horizontally, the nodes and identifiers are stored in the following order: [ajs], [bd], [bie], [18], [amant], [11], [o], [ei], [56], [ning], [38], [et], [mester], [77], [an], [dfl], [26], [ord], [63], [ey], [0]. Vertically, the nodes and identifiers are stored in the following order: [ajs], [bd], [o], [et], [bie], [amant], [ei], [mester], [an], [18], [11], [56], [ning], [77], [dfl], [38], [26], [ord], [ey], [63], [0].

The hybrid embodiment 500 has some of the same disadvantages as the vertical embodiment 400, but not as pronounced. For example, in order to find the data item "joining", ten nodes must be read. In addition, there are two types of nodes, rather than one, and the software needed to implement the hybrid embodiment would be correspondingly more complex.

Like the vertical embodiment, because the horizontal embodiment is more efficient in most applications, it is preferred over the hybrid embodiment. Consequently, the hybrid embodiment has not been implemented. It, too, is included here to show that the embodiment, although possible, is not a practical application of the present invention.

Thus it has been shown and described a data structure which satisfies the objects set forth above.

Since certain changes may be made in the present disclosure without departing from the scope of the present invention, it is intended that all matter described in the foregoing specification and shown in the accompanying drawings be interpreted as illustrative and not in a limiting sense.

What is claimed is:

1. A data structure adapted for use with computation means for storing data items in a data base, each of said data items being composed of at least one data segment, each of said data segments being composed of at least one data unit including a first data unit, said data structure comprising:

(a) a linear stream of nodes, each of said nodes containing a data segment, said nodes including predecessor nodes and successor nodes and being related by location in said stream;

(b) progressions of said nodes in said stream from a root alternate list member node to a leaf node corresponding to progressions of data segments of said data items;

(c) said progressions being traversed from node information included in each of said nodes, said node information including a size value indicating the number of said data units in said node, a leaf node flag indicating if said node is a leaf node, and a terminal alternate flag indicating if said node is an alternate list terminal node;

(d) each of said progressions being associated with at least one identifier that is adapted to reference at least one object external to said stream;

(e) selected data segments of different data items in selected progressions of successor nodes being different for different data items; and

(f) selected data segments of different data items in selected predecessor nodes being common to said different data items.

2. The data structure of claim 1 wherein:

(a) said computation means includes a plurality of memory blocks;

(b) said stream is distributed among said memory blocks;

(c) said node information includes a pointer flag indicating if said node includes a memory block pointer that points to a memory block; and

(d) said node information includes a memory block pointer if said pointer flag is set.

3. The data structure of claim 2 wherein:

(a) each of said memory blocks contains one or more subtrees of said stream of nodes;

(b) all of said subtrees of one memory block are pointed to from nodes of a single alternate list;

(c) each of said subtrees of said one memory block are pointed to from a different node of said single alternate list; and

(d) said each of said subtrees of said one memory block contains a copy of said first data unit from said different node;

(e) whereby when said memory block pointer is followed from a node of an alternate list, a subtree continuing said progression of nodes is found by matching said first data unit with said copy of said first data unit.

4. The data structure of claim 3 wherein:

(a) each of said subtrees includes a subtree root alternate list;

(b) said single alternate list is a subtree root alternate list; and

(c) each of said subtrees includes a back pointer to said memory block containing said single alternate list;

(d) whereby said progressions of nodes are traversed in reverse by following said back pointer and matching said copy of said first data unit with said first data unit of each of said nodes of said single alternate list to determine which one of said nodes of said single alternate list was traversed to reach said subtree containing said copy of said first data unit.

5. The data structure of claim 3 wherein:

(a) said data structure further includes a memory block stack;

(b) said memory block stack being comprised of memory block information of said memory blocks traversed as said progressions of nodes are traversed that is pushed on said memory block stack;

(c) each of said subtrees includes a subtree root alternate list; and

(d) said single alternate list is a subtree root alternate list;

(e) whereby said progressions of nodes can be traversed in reverse by popping said memory block information from said stack and matching said copy of said first data unit with said first data unit of each of said nodes of said single alternate list to determine which one of said nodes of said single alternate list was traversed to reach said subtree containing said copy of said first data unit.

6. The data structure of claim 1 wherein each of said progressions of nodes is associated with one identifier, said identifier being included in said leaf node of said each of said progression of nodes.

7. The data structure of claim 1 wherein said at least one identifier is stored in said stream immediately after said leaf node of each of said progressions of nodes.

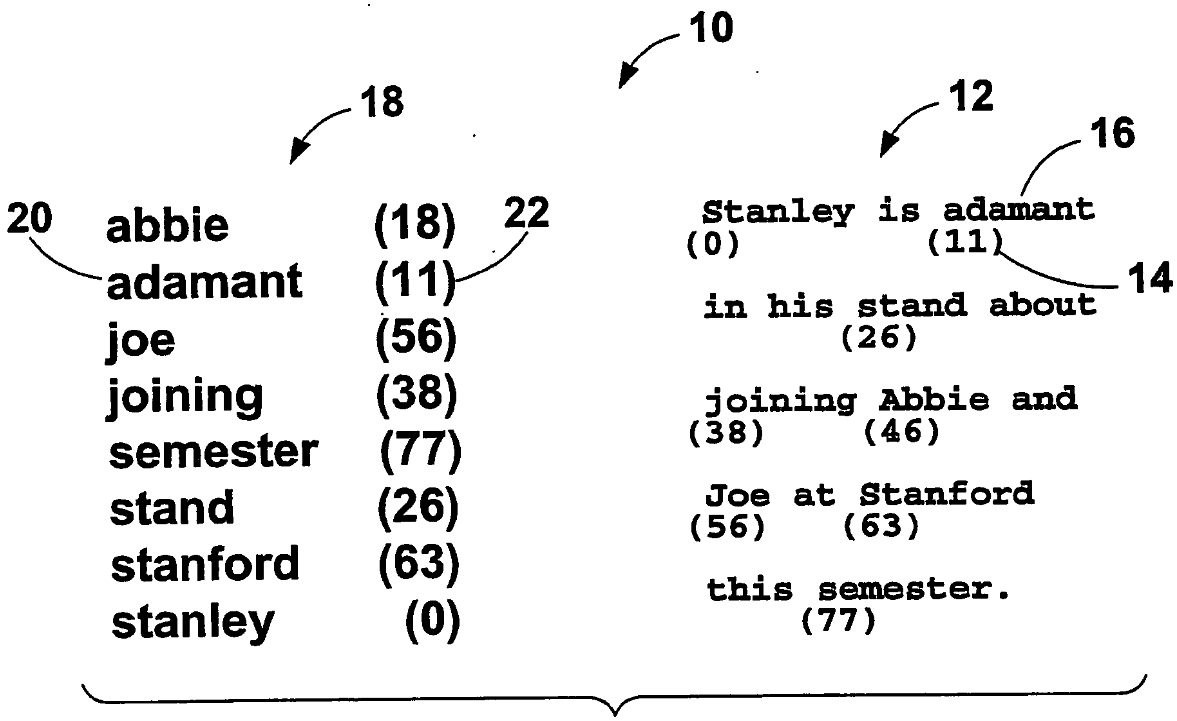

10
18
12
16
20
22
14
abbie (18)
adamant (11)
joe (56)
joining (38)
semester (77)
stand (26)
stanford (63)
stanley (0)
Stanley is adamant
(0) (11)
in his stand about
(26)
joining Abbie and
(38) (46)
Joe at Stanford
(56) (63)
this semester.
(77)

Fig. 1

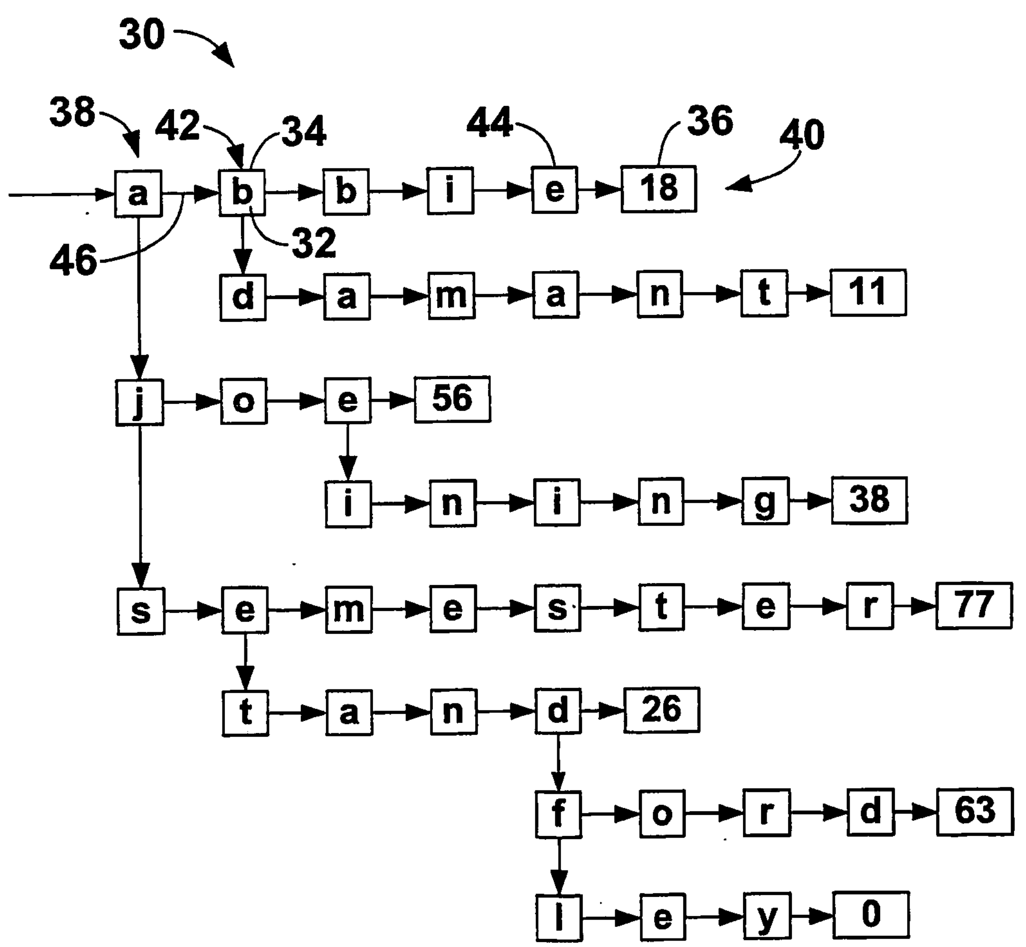

30
38
42
34
44
36
40
46
32
a b b i e 18
d a m a n t 11
j o e 56
i n i n g 38
s e m e s t e r 77
t a n d 26
f o r d 63
l e y 0

Fig. 2

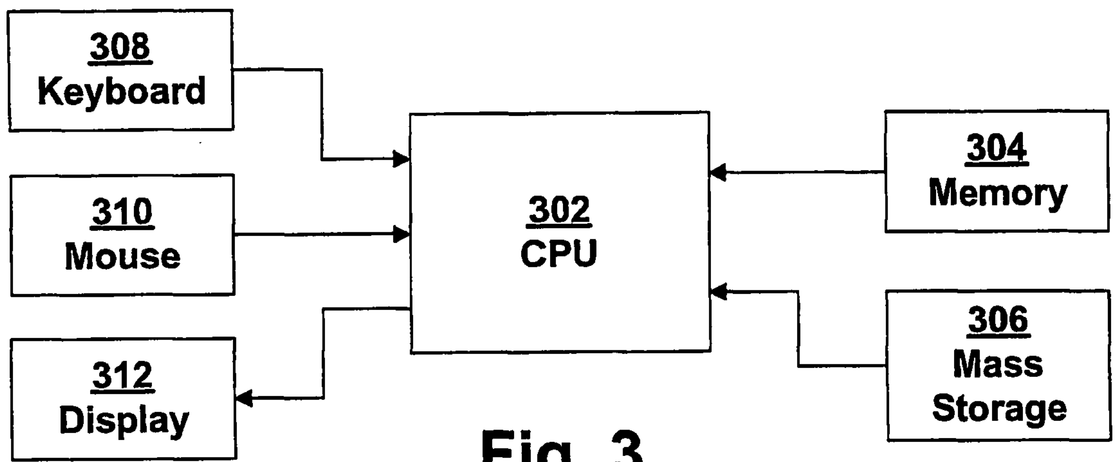

308
Keyboard
310
Mouse
312
Display
302
CPU
304
Memory
306
Mass
Storage

Fig. 3

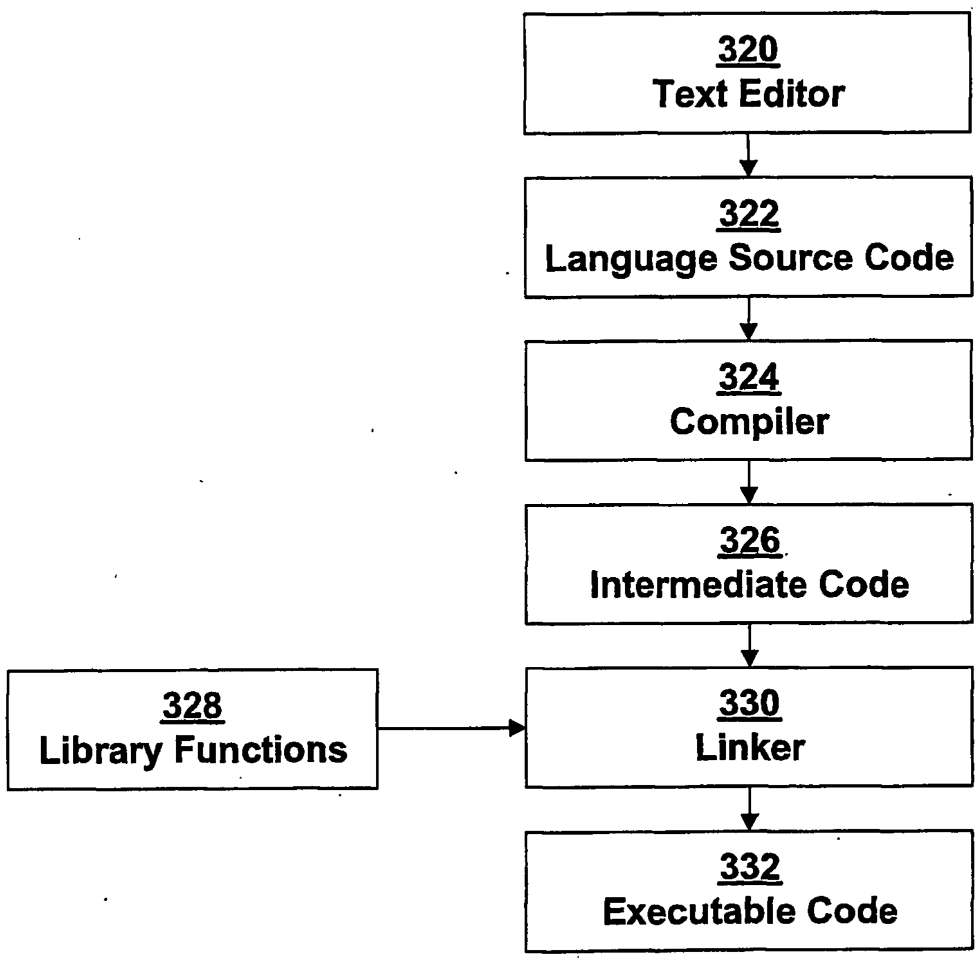

320
Text Editor
322
Language Source Code
324
Compiler
326
Intermediate Code
328
Library Functions
330
Linker
332
Executable Code

Fig. 4

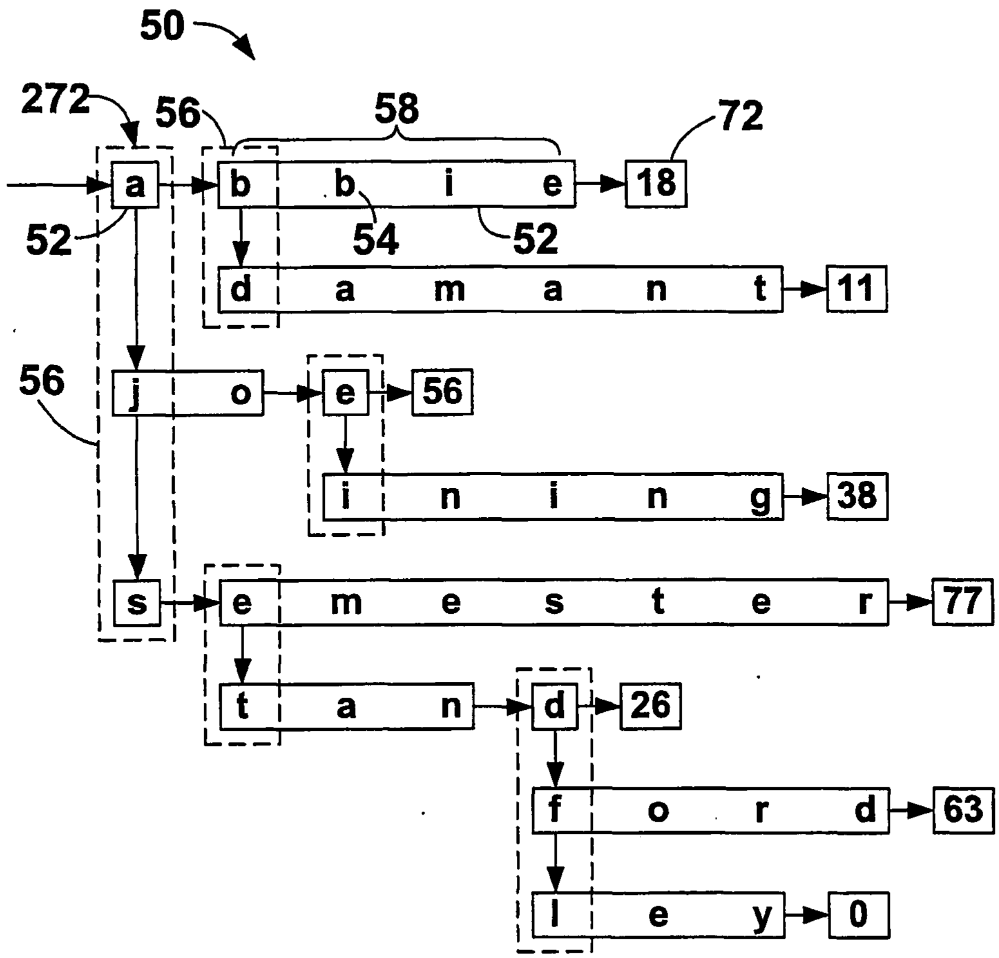



Fig. 5

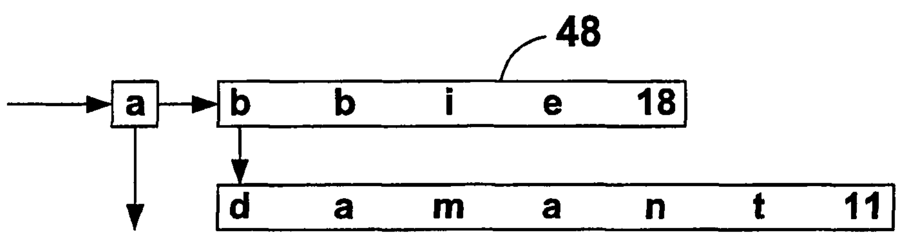



Fig. 6

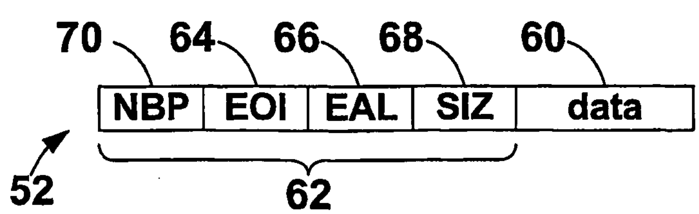



Fig. 7

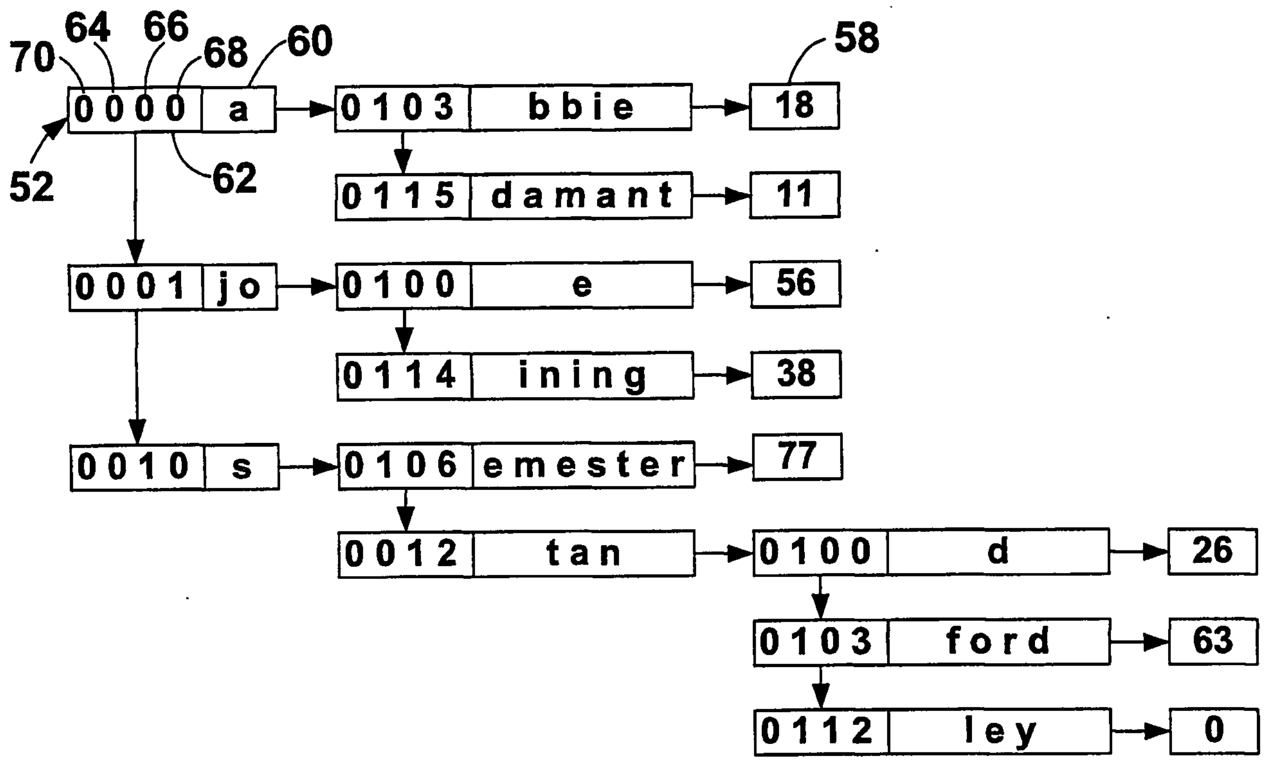

70
64
66
68
60
58
0000 a
0103 bbie
18
0115 damant
11
52
62
0001 jo
0100 e
56
0114 ining
38
0010 s
0106 emester
77
0012 tan
0100 d
26
0103 ford
63
0112 ley
0

Fig. 8

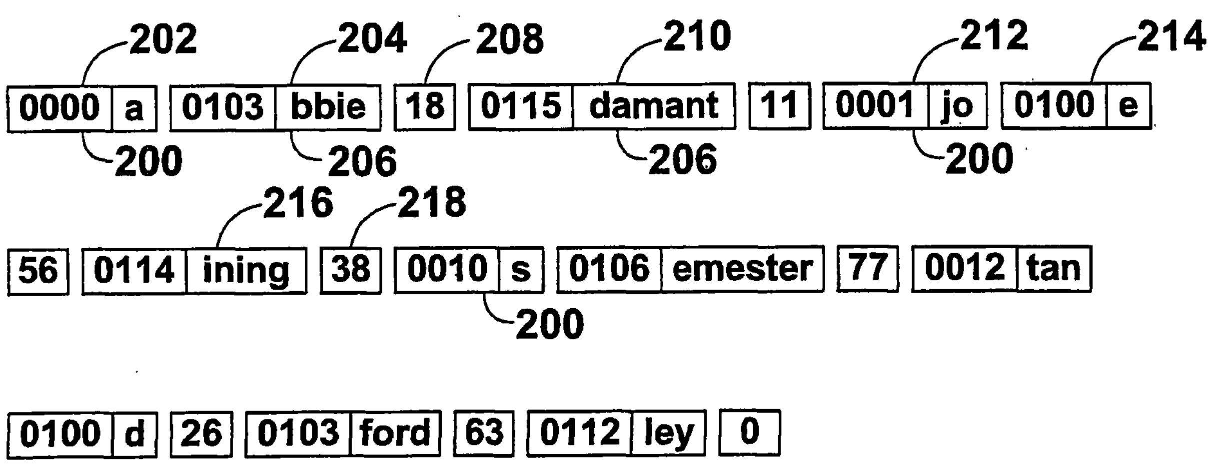

202
204
208
210
212
214
0000 a
0103 bbie
18
0115 damant
11
0001 jo
0100 e
200
206
206
200
216
218
56
0114 ining
38
0010 s
0106 emester
77
0012 tan
200
0100 d
26
0103 ford
63
0112 ley
0

Fig. 9

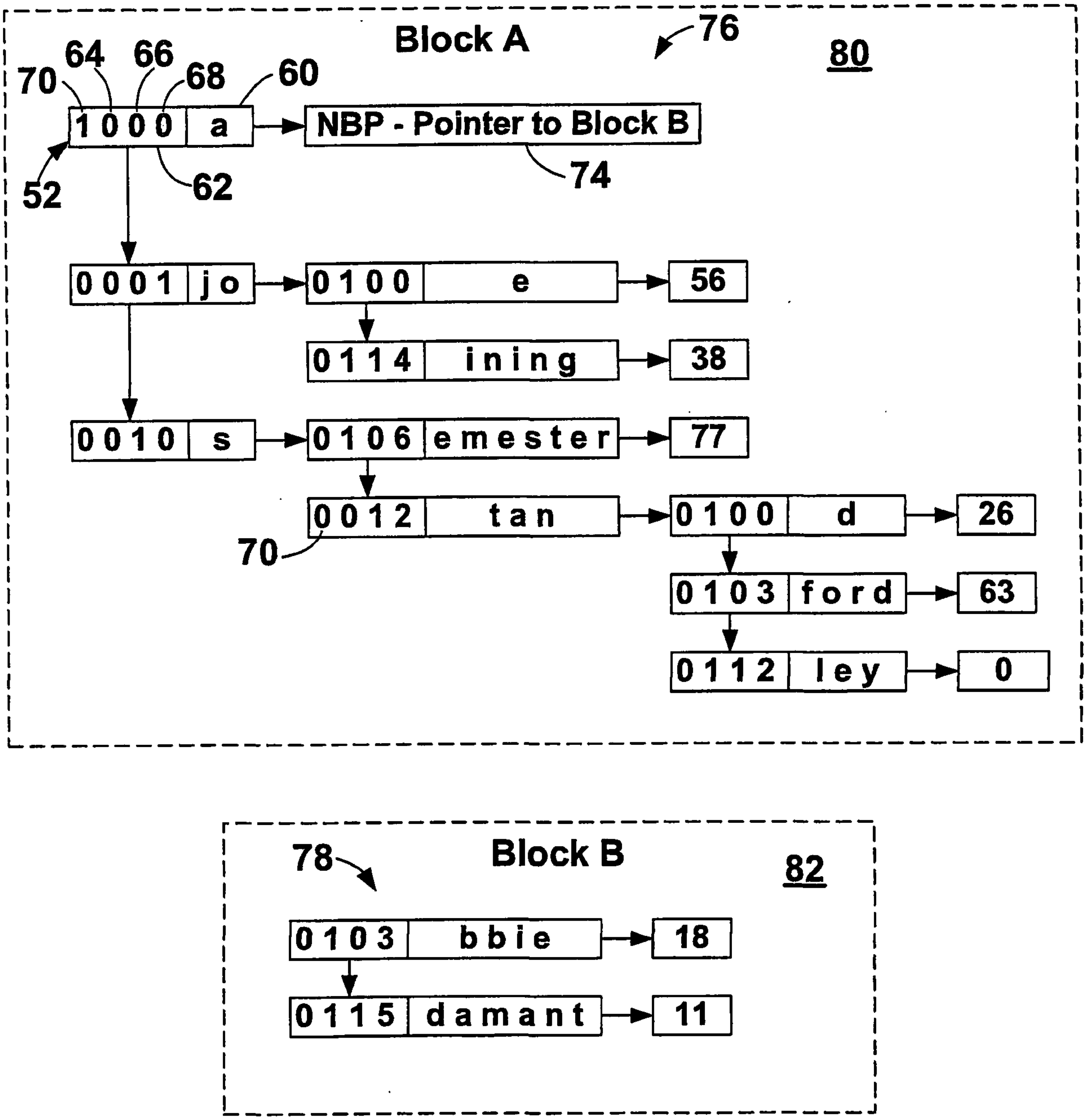
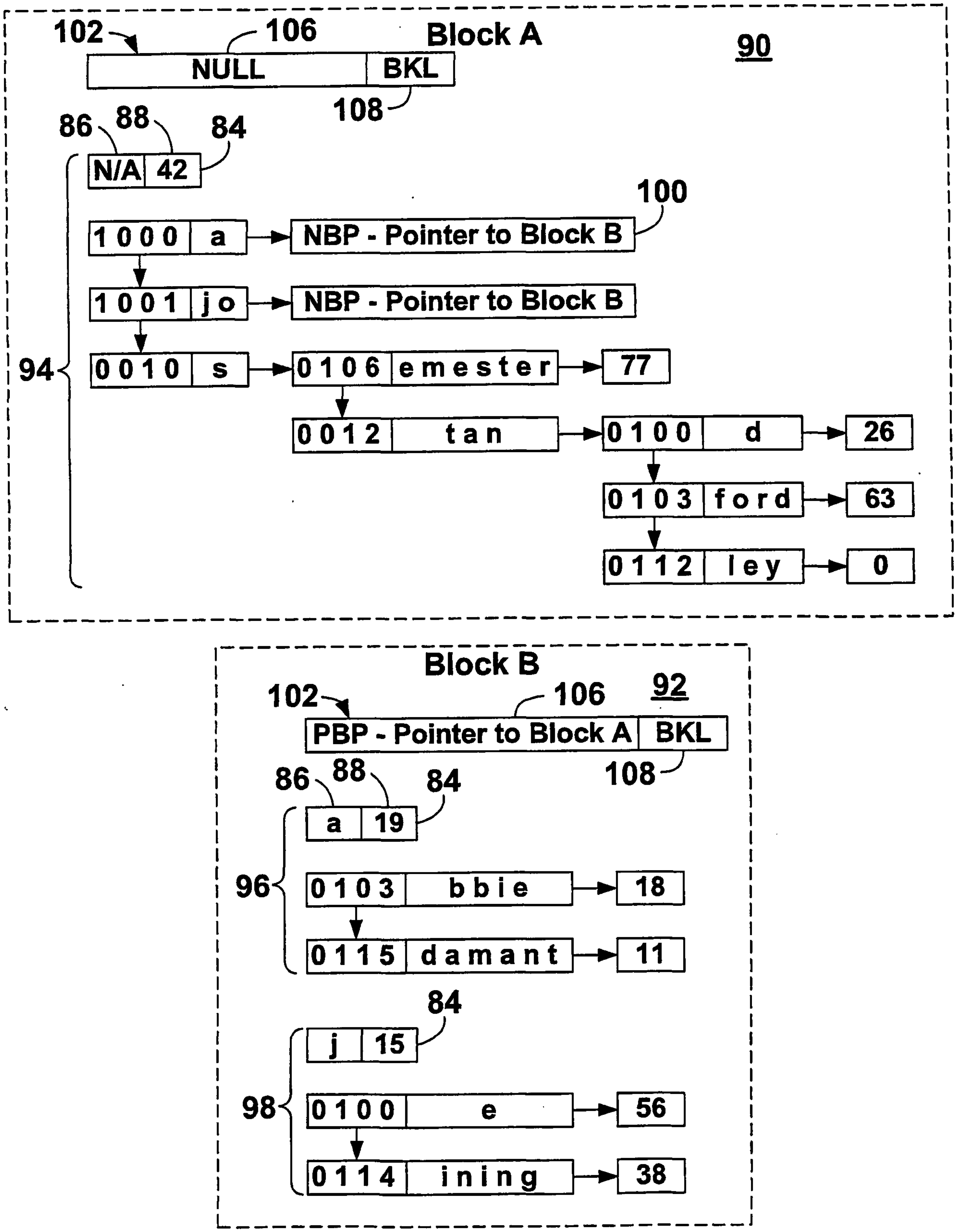

Block A
76
80
70
64
66
68
60
1000
a
NBP - Pointer to Block B
52
62
74
0001
jo
0100
e
56
0114
ining
38
0010
s
0106
emester
77
0012
tan
0100
d
26
70
0103
ford
63
0112
ley
0
Block B
78
82
0103
bbie
18
0115
damant
11

Fig. 10



Block A
90
102
106
NULL
BKL
108
86
88
84
N/A
42
100
1 0 0 0
a
NBP - Pointer to Block B
1 0 0 1
j o
NBP - Pointer to Block B
94
0 0 1 0
s
0 1 0 6
e m e s t e r
77
0 0 1 2
t a n
0 1 0 0
d
26
0 1 0 3
f o r d
63
0 1 1 2
l e y
0
Block B
102
106
92
PBP - Pointer to Block A
BKL
108
86
88
84
a
19
96
0 1 0 3
b b i e
18
0 1 1 5
d a m a n t
11
84
j
15
98
0 1 0 0
e
56
0 1 1 4
i n i n g
38


Fig. 11

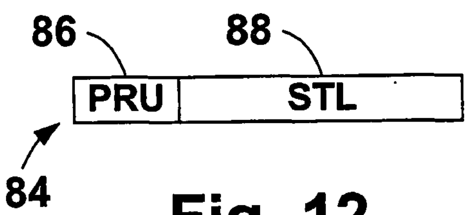



Fig. 12

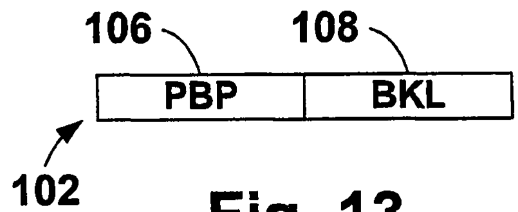



Fig. 13

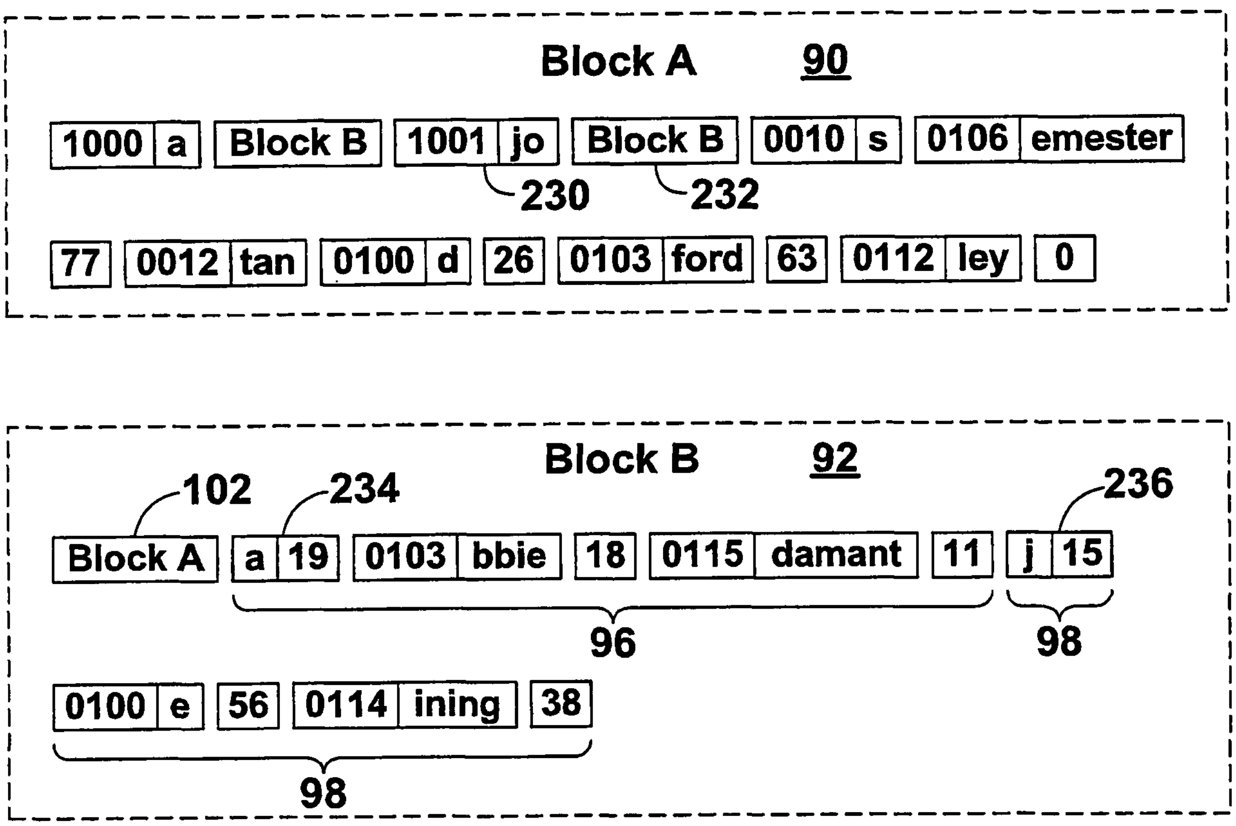



Fig. 14

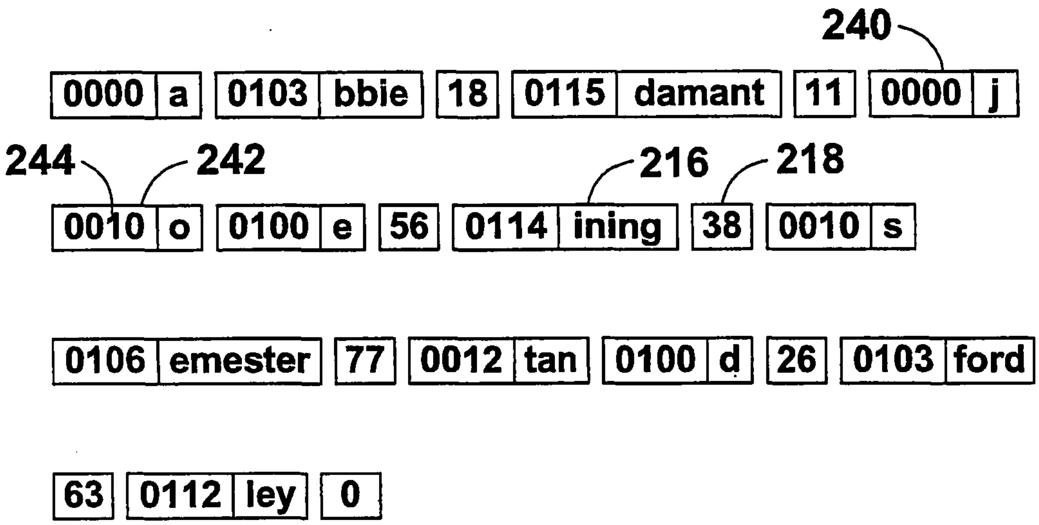

240
0000 a
0103 bbie
18
0115 damant
11
0000 j
244
242
216
218
0010 o
0100 e
56
0114 ining
38
0010 s
0106 emester
77
0012 tan
0100 d
26
0103 ford
63
0112 ley
0

Fig. 15

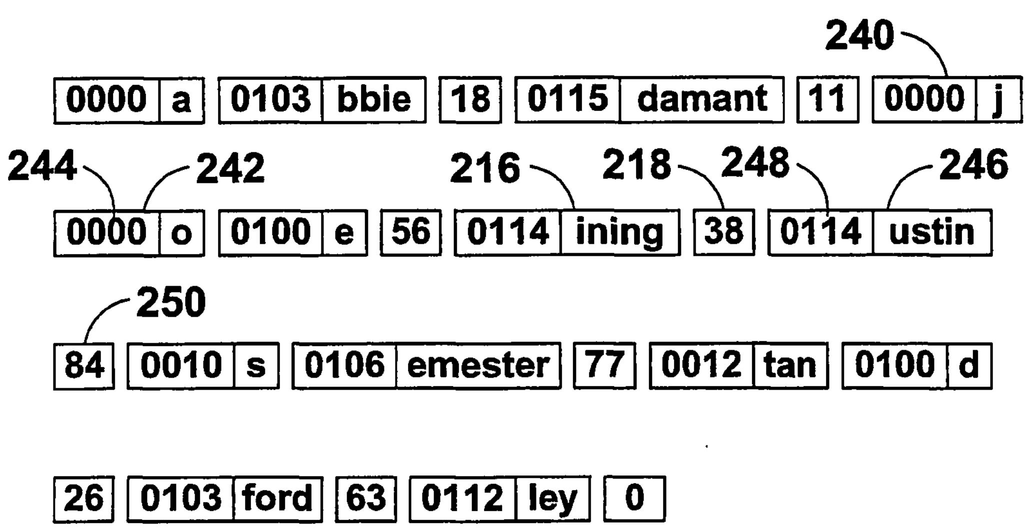

240
0000 a
0103 bbie
18
0115 damant
11
0000 j
244
242
216
218
248
246
0000 o
0100 e
56
0114 ining
38
0114 ustin
250
84
0010 s
0106 emester
77
0012 tan
0100 d
26
0103 ford
63
0112 ley
0

Fig. 16

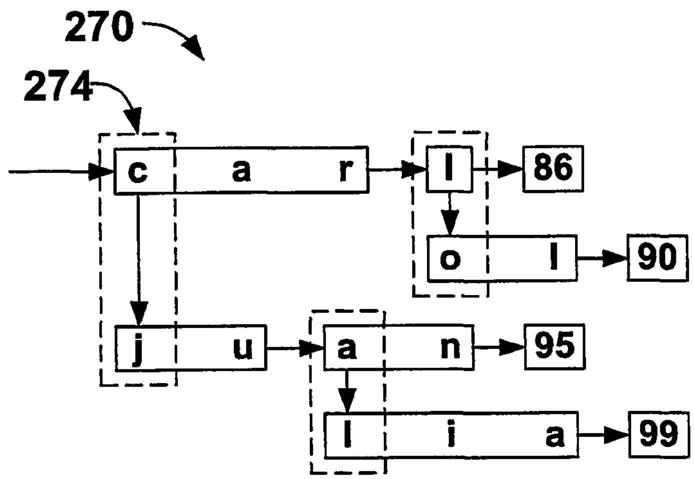



Fig. 17

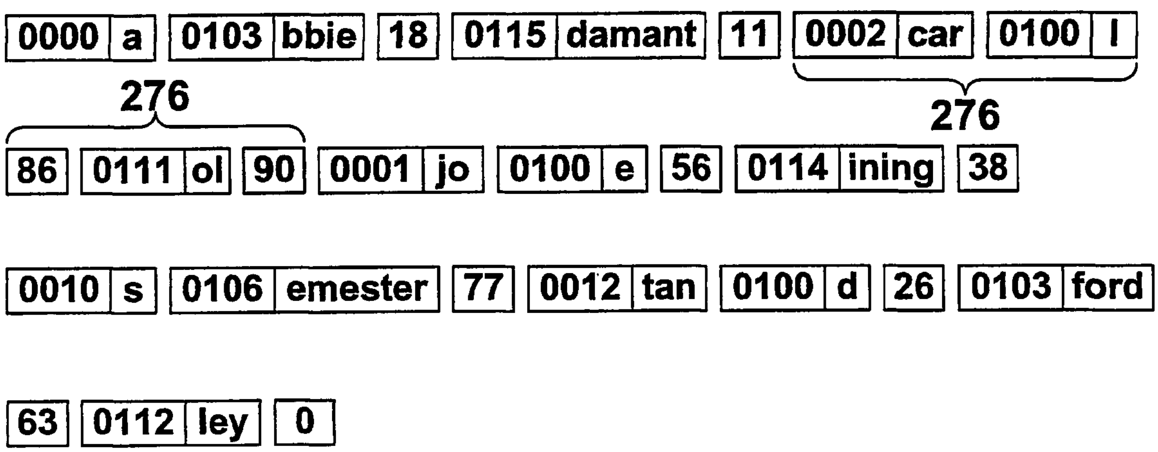



Fig. 18

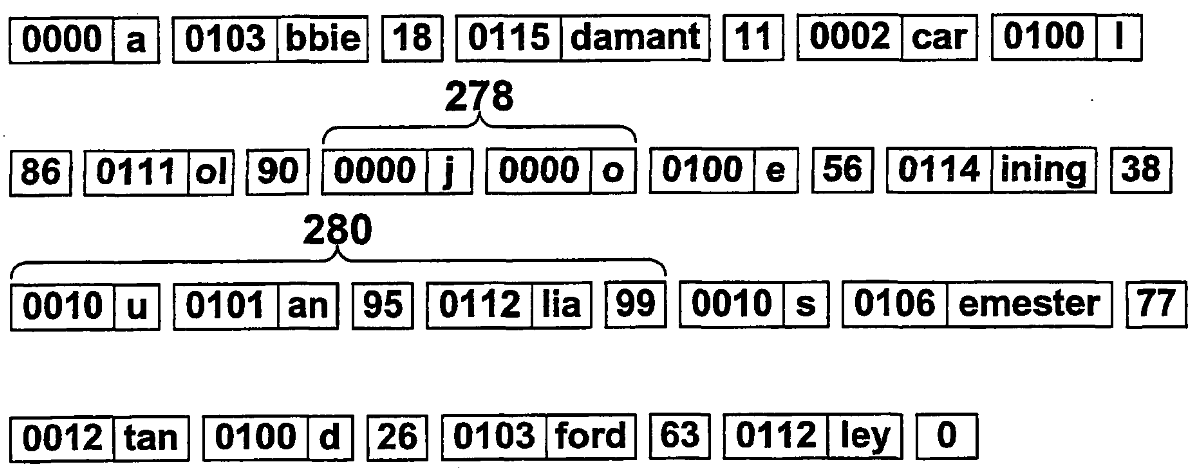
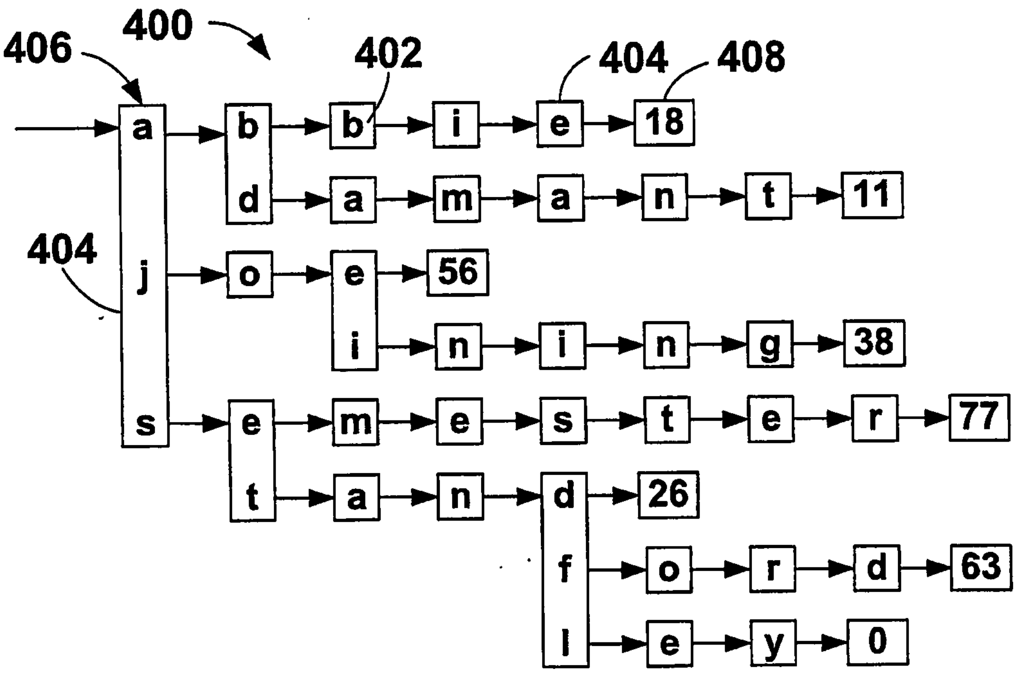
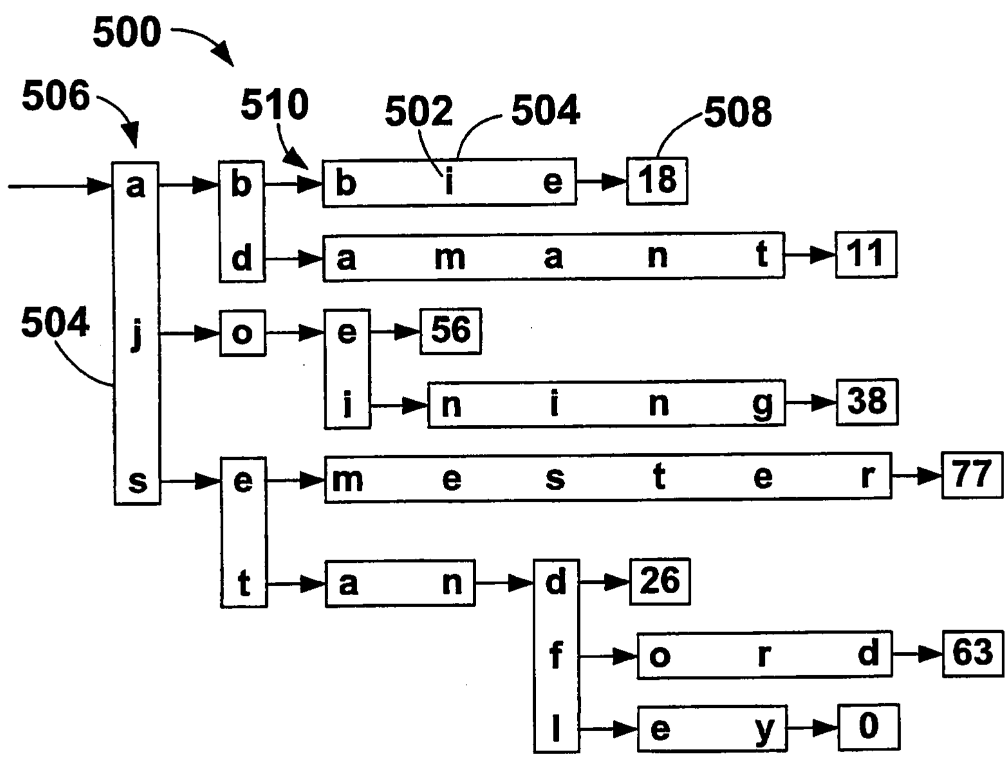



Fig. 19



400
406
402
404
408
a
b
b
i
e
18
d
a
m
a
n
t
11
404
j
o
e
56
i
n
i
n
g
38
s
e
m
e
s
t
e
r
77
t
a
n
d
26
f
o
r
d
63
l
e
y
0


Fig. 20



500
506
510
502
504
508
a
b
b i e
18
d
a m a n t
11
504
j
o
e
56
i
n i n g
38
s
e
m e s t e r
77
t
a n
d
26
f
o r d
63
l
e y
0


Fig. 21

# INTERNATIONAL SEARCH REPORT

International application No.
PCT/US00/34523

**A. CLASSIFICATION OF SUBJECT MATTER**

IPC(7) : G06F 17/00, 7/00
US CL : 707/100

According to International Patent Classification (IPC) or to both national classification and IPC

**B. FIELDS SEARCHED**

Minimum documentation searched (classification system followed by classification symbols)
U.S. : 707/100, 101, 102, 1, 2

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
EAST

**C. DOCUMENTS CONSIDERED TO BE RELEVANT**

| Category * | Citation of document, with indication, where appropriate, of the relevant passages | Relevant to claim No. |
|---|---|---|
| A | US 5,737,732 A (GIBSON et al) 07 April 1998 (07.04.1998), ALL | 1-7 |
| A | US 5,283,894 A (DERAN) 01 February 1994 (01.02.1994), ALL | 1-7 |
| A, P | US 6,067,574 A (TZENG) 23 May 2000 (23.05.2000), ALL | 1-7 |

☐ Further documents are listed in the continuation of Box C. ☐ See patent family annex.

* Special categories of cited documents:

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier application or patent published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search

Date of mailing of the international search report
21 MAR 2001

Name and mailing address of the ISA/US
Commissioner of Patents and Trademarks
Box PCT
Washington, D.C. 20231
Facsimile No. (703)305-3230

Authorized officer
Thomas Black James R. Matthews
Telephone No. 305-9000